Timely and cost-efficient data exploration through adaptive tuning

Thèse N° 9636

Présentée le 18 octobre 2019

à la Faculté informatique et communications
Laboratoire de systèmes et applications de traitement de données massives
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Matthaios Alexandros OLMA

Acceptée sur proposition du jury

Prof. K. Aberer, président du jury
Prof. A. Ailamaki, directrice de thèse
Prof. S. Idreos, rapporteur
Dr S. Kandula, rapporteur
Prof. R. Guerraoui, rapporteur

2019

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

While experiencing happiness,
we have difficulty in being conscious of it.
Only when happiness is past and we look back on it
we do realize — how happy we had been.

— Nikos Kazantzakis

To my amazing family and my more than awesome friends.

Acknowledgements

In this short section, I would like to thank all the people which helped me put together this thesis, and shared with me both the difficult and joyful moments of this journey.

First and foremost, I would like to thank my advisor, Anastasia Ailamaki, for her guidance, advice and for believing in me. Natassa as a supervisor has shaped me as a researcher and as an engineer. Natassa is undeniably an amazing researcher; insightful, hard-working and always on target. However, two characteristics make her extraordinary: her love for teaching and her care over her students. Natassa invests time and effort to guide students to pursue excellence and find happiness. These characteristics make the DIAS group an amazing place to study, learn and a safe harbour for emerging scientists and people.

I would also like to thank the members of my thesis committee, for their participation as well as their insightful and constructive comments which shaped this thesis. Specifically, I would like to thank Srikanth Kandula, who acted as my mentor during my internship at Microsoft Research which heavily influenced the second part of this thesis, Stratos Idreos whose own work has influenced immensely this thesis, Rachid Guerraoui who has provided feedback to me ever since my first year at EPFL, and Karl Aberer, who accepted to act as the thesis jury president.

However, this thesis is the collaborative work of an amazing group of people I had the fortune and honor to work with. Thomas Heinis, Farhan Tauheed, Ioannis Alagiannis, Pinar Tozun, Manos Athanassoulis, Iraklis Psaroudakis, Manos Karpathiotakis, Danica Porobic, Raja Appuswamy, Odysseas Papapetrou, Stella Giannakopoulou, Periklis Chrysogelos, Panos Sioulas each and everyone has influenced me greatly in the way I work and think; I thank you all for our collaboration. I would like to thank the Master students and interns, that I had the happiness to work with. Antonis Anagnostou, Junze Bao, Ergys Dona, Mark Jolles, Moritz Raho, Shahnur Isgandarli, Ekaterina Dopiro and Kostantinos Tsitsimpikos for their enthousiasm and hard work. Furthermore, apart from the immediate collaborators, the DIAS laboratory has an amazing culture which made everyday life an joyful experience. Adrian, Angelos, Aunn, Ben, Bikash, Cesar, Christina, Danica, Darius, Diane, Dimitra, Eleni, Erietta, Erika, Farhan, Fotini, George, Ioannis, Iraklis, Konstantinos, Lionel, Manos A., Manos K., Mirjana, Odysseas, Panagiotis, Periklis, Pinar, Radu, Raja, Rakesh, Renata, Satya, Snow, Stella, Tahir, Thomas, Utku, and Viktor were always happy to provide feedback, chat about research and relax. I would

also like to thank my awesome neighbors from the DATA lab, Amir, Mohammed and Yannis for making the corridor a lively place. I want to express my immense gratitude to a special group of people that stood for me more than family in this journey. There are no words to express my gratitude to Manos Karpathiotakis, he is an amazing researcher, engineer and more importantly friend, his patience, knowledge and reason have made their permanent mark on me. Ioannis Alagiannis without whose guidance, experience and understanding this thesis would fail. Manos Athanassoulis who took me aside after my first presentation in the lab and still guides me ever since. Raja Appuswamy who is the best officemate I could ever ask for, an amazing researcher, engineer and beer connaisseur. Stella Gianakopoulou, who apart from being the best TA in EPFL and Spark expert, is an awesome collaborator and friend. Eleni and Mira who were never scared. Danica and Pinar whose advice on anything was golden. Iraklis for his amazing awesomeness. I thank you all for your patience and help.

Lausanne is a beautiful city, however, the people I met here made it the best place in the world. I would like to thank Manos K., Eleni, Stella, Iris, Lorenzo, Vasilis Trig., Aggelos B., Stelios, Panos Smeros, Panos Sioulas, Loukia, Ioannis, Manos A. for being my family away from home, for our amazing philosophical discussions and invigorating moments. I would like to thank my girlfriend Sylvia for keeping me alive and bearing with me during the final stretch of the PhD. I am grateful to Pavlos and Christos for being my go-to people for things I don't understand. I would like to thank Katerina, Javier, Nadia, Christina, Antriani, and Nathalie that made any sad moments instantly happy. Sergio, Fred, George Chatzopoulos, George Prekas, Vassilis Agrafiotis, Vassilis Koukoulomatis, Marios, Matthildi, Maria, Iraklis, Christos, Natassa, Evi, Giannis, Apostolis, Dejan and many more – thank you all for being there.

My parents and my brother have been supporting me throughout my studies. My parents met pursuing their PhD (and claim it was the best time of their lives) and advised me to pursue a PhD as well. Thank you very much for your advice and support. One of my proudest moments was when I attended my brother's PhD defense, I hope I was as supportive as you were.

This research has been supported by grants from the School of Computer and Communication Sciences, EPFL, the European Union Seventh Framework Programme (ERC-2013-CoG), under grant agreement no 617508 (ViDa), the EU FP7 Collaborative project Grant No 317858 (BigFoot), the EU Horizon 2020 research and innovation programme Grant No 650003 (Human Brain project), and the EU Horizon 2020 research and innovation programme under Grant No 825041.

Lausanne, 18 July 2019 M. O.

Abstract

Modern applications accumulate data at an exponentially increasing rate and traditional database systems struggle to keep up. Decision support systems used in industry, rely heavily on data analysis, and require real-time responses irrespective of data size.

To offer real-time support, traditional databases require long preprocessing steps, such as data loading and offline tuning. Loading transforms raw data into a format that reduces data access cost. Through tuning, database systems build access paths (e.g., indexes) to improve query performance by avoiding or reducing unnecessary data access. The decision on what access paths to build depends on the expected workload, thus, the database system assumes knowledge of future queries. However, decision support systems and data exploration applications have shifting requirements. As a consequence, an offline tuner with no *a priori* knowledge of the full workload is unable to decide on the optimal set of access paths. Furthermore, access path size increases along with input data, thus, building precise access paths over the entire dataset limits the scalability of databases systems.

Apart from long database pre-processing, offering efficient data access despite increasing data volume becomes harder due to hardware architectural constraints such as memory size. To achieve low query latency, modern database systems store data in main memory. However, there is a physical limit on main memory size in a server. Therefore, applications must trade memory space for query efficiency.

To provide high performance efficiency, irrespective of dataset growth and query workload, a database system needs to (i) shift the decision of tuning from off-line to query-time, (ii) enable the query engine to exploit application properties in choosing fast access paths, and (iii) reduce the size of access paths to limit storage cost.

In this thesis, we present techniques for query processing that are adaptive to workload, application requirements, and available storage resources. Specifically, to address dynamic workloads, we turn access path creation into a continuous process which fully adapts to incoming queries. We assign all decisions on data access and access path materialization to the database optimizer at query time, and enable access path materialization to take place as a by-product of query execution, thereby, removing requirements for long offline tuning processing steps. Furthermore, we take advantage of application characteristics (precision requirements, resource availability) and we design a system which can adaptively trade precision and resources for performance. By combining precise and approximate access paths, the database system reduces query response time and minimizes resource utilization. Approximate access paths (e.g., sketches) require less space in comparison to their precise counterparts, and offer

lower access time.
By improving data processing performance while reducing storage requirements through (i) adaptive access path materialization and (ii) using approximate and space-efficient access paths when appropriate, our work minimizes data access cost and provides real-time responses for data exploration applications irrespective of data growth.

Résumé

Les applications modernes accumulent les données à un rythme augmentant de façon exponentiel et les systèmes de base de données traditionnels n'arrive pas à le suivre. Les systèmes d'aide à la décision utilisés dans l'industrie reposent largement sur l'analyse de données, et nécessitent des réponses en temps réel, indépendamment de leur taille.

Pour offrir un support en temps réel, les bases de données traditionnelles nécessitent de longues étapes de prétraitement, telles que le chargement de données et des réglages fin de leurs paramètres. Le chargement transforme les données brutes en un format de données qui réduit les coûts d'accès à celles-ci pour la base de données. Grâce aux réglages fin, les bases de données créent un ensemble de chemins d'accès (par exemple, des index) qui améliorent les performances des requêtes en réduisant l'accès aux données. La décision sur les chemins d'accès à construire dépend de la charge de travail future, ce qui implique des hypothèses sur les requêtes à venir. Cependant, les systèmes d'aide à la décision et les applications d'exploration de données ont des exigences changeantes. En conséquence, un optimisateur hors ligne sans aucune connaissance a priori de la charge de travail complète est incapable de décider de l'ensemble optimal des chemins d'accès. En outre, la taille des chemins d'accès augmente avec les données en entrée, la construction de chemins d'accès précis sur l'ensemble du jeu de données limite la capacité à monter en charge des systèmes de bases de données.

En dehors des longues opérations de prétraitement, offrir des accès efficaces aux données, et ce malgré l'augmentation de leur volume, devient de plus en plus difficile en raison des contraintes matérielles et architecturales, telles que la taille de la mémoire et la bande passante des bus de transfert de données. Les systèmes de base de données modernes minimisent la latence des requêtes en stockant les données en mémoire principale. Cependant, il existe une limite physique sur la taille de la mémoire principale dans un serveur. Pour remédier à cela, certains systèmes stockent des données dans la mémoire de plusieurs serveurs et partage leurs ressources, ce qui entraine des transferts de données sur le réseau. En utilisant cette architecture, la performance des applications de traitement de données distribuées est limitée par la latence d'accès aux données résidant sur de la mémoire distante.

Pour fournir de hautes performances de manières efficaces, et ce indépendamment de la croissance du jeu de données ou de la charge de travail des requêtes, un système de base de données doit (i) faire passer la décision des réglages fin hors ligne au moment de la requête, (ii) permettre au moteur de requête d'exploiter les propriétés de l'application pour choisir des

chemins d'accès rapides et peu encombrants.
Dans cette thèse, nous présentons des techniques de traitement de requêtes qui sont : adaptable à la charge de travail, aux exigences des applications ainsi qu'à celles des architectures des systèmes distribués. Plus précisément, pour traiter les charges de travail dynamiques, nous transformons la création des chemins d'accès en un processus continu qui s'adapte parfaitement aux requêtes en entrée. Nous attribuons toutes les décisions concernant l'accès aux données et la matérialisation des chemins d'accès à l'optimisateur de la base de données, au moment de la requête, ainsi la matérialisation peut avoir lieu comme un sous-produit de l'exécution de la requête ; supprimant par conséquent le besoin pour un long traitement hors ligne pour effectuer les réglages fins. De plus, nous tirons parti des caractéristiques de l'application (contraintes de précision, disponibilité des ressources) et nous concevons un système qui peut échanger la précision et des ressources pour la performance. En combinant chemins d'accès précis et approximatifs, le système de base de données peut réduire le temps de réponse des requêtes tout en minimisant l'utilisation des ressources. Les chemins d'accès approximatifs (par exemple, les esquisses) nécessitent moins d'espace comparés à leurs homologues précis, et offrent un temps d'accès constant.
En améliorant les performances du traitement des données tout en réduisant les exigences de stockage via (i) la matérialisation des chemins d'accès adaptative, et (ii) l'utilisation des chemins d'accès approximatifs, lorsque cela est possible, notre travail minimise les coûts d'accès aux données et fournit des réponses en temps réel pour les applications d'exploration des données, et ce indépendamment de la croissance de celles-ci.

Contents

List of Figures

List of Tables

1 Introduction

An increasing number of applications in various domains generate and collect massive amounts of data at a rapid pace. Research fields and applications (e.g., network monitoring, sensor data management, clinical studies, etc.) emerge and require broader data analysis functionality to rapidly gain deeper insights from the available data. Despite the advancement of computer hardware and data management technology during the past years, analyzing and understanding all available data is infeasible in practice due to the data explosion of the last decade. The increased complexity of workloads, with respect to predictability and computation, combined with the ever-increasing datasets, pose challenges to traditional data management solutions.

1.1 The need for data exploration

There is an increasing trend appearing in applications, ranging from information retrieval to human-computer interaction and visualization communities, in moving beyond the traditional query-browse-refine model supported by database systems, and towards support for human intelligence amplification and information understanding. To enhance user experience, such applications employ data exploration [64]. In data exploration scenarios, users explore available data iteratively for actionable information, trying to assess the problem space before making a decision [27, 28, 53, 64, 77]. Typically, each query is formulated based on preceding queries and their results. The performance of a data exploration application is determined by its ability to process data in a timely manner maximally utilizing available computational resources [16].

- **Interactivity.** For data scientists low response time is known to increase productivity. A recent study [83] shows that even small delays (more than 500ms) significantly decrease user's activity level, dataset coverage and insight discovery. Thus, low response times are of high importance. Furthermore, extracting useful information from data the moment it is available is important, as data could lose its value with time.

- **Resource utilization.** Due to the increasing data processing work, many companies and organizations move their data analytics workloads to the cloud, where they are charged based on resource utilization. In order to reduce the cost of data processing, applications must minimize resource utilization, take full advantage of available resources, and be elastic to the available resources.

Traditional DBMS approaches are designed based on a predefined information demand, meaning that the analytical queries they support are fixed in advance to what their predefined setup allows. Essentially, a DBMS is designed to efficiently answer a specific set of queries. However, in exploratory search, the information demand is unknown a priori, and the goal of the search is also to enable the discovery of aspects unknown during design time. The dynamic nature of exploratory workloads, with unpredictable query patterns, poses a new challenge to database systems, since optimal physical design becomes a moving target rather than a one time investment. With the query workload and data accessed being highly diverse, complete physical design re-calibration may turn out to be an unprofitable investment for future queries.

1.2 The effects of increasing data volume

Data collections are ever-increasing in size. To illustrate, the amounts of machine-generated data are, per one estimate [66], increasing by a factor of 44 each year, and are estimated to reach the value of 35 ZB by 2020. Given the data sizes involved, any transformation, copying, and preparation steps over the data introduces substantial delays before the data can be utilized and queried [3, 10, 65]. However, recent studies of data analysis workloads show that typically only a small subset of the data is relevant and ultimately used by exploratory workloads [86]. Furthermore, due to the growth of data volume, databases store data in multiple formats and on hardware with varying access latencies. Furthermore, any investments toward auxiliary data structures improving data access become increasingly expensive both in execution time and storage space. In conjunction with the costly construction, when going through never-before-seen data, no assumptions can be made about the data or the queries. A user may become interested in different value ranges and/or attributes. Workload shifts may nullify investments towards auxiliary data structures because predicting in which areas of the dataset to invest is non-trivial. Furthermore, conventional approaches require building full and precise indexes whose size increases linearly with the size of data thus require increasingly more space. Thus, the ever growing datasets pose two challenges on conventional DBMS approaches:

- **Filtering relevant data.** To enable high performance analytics, conventional DBMS physical design approaches build and maintain auxiliary structures to reduce data access. However, the size of such auxiliary structures increases linearly with the size of data thus creating scalability problems.

- **Storing and Accessing data.** Modern DBMS store increasingly more data in memory to have low-latency access. However, the use of high-density DRAM chips to sustain the ever-growing needs for memory increases energy costs.

1.3 Thesis statement and Contributions

In this thesis we demonstrate that the conventional database physical design and data access approaches are insufficient to cover the need of modern analytical workloads. The current paradigm of fully loading the data, tuning the system with complete and precise auxiliary structures which aims to provide the desired interactive nature of analysis, has become a bottleneck. Database systems must fully adapt to the dataset, workload characteristics, user requirements and modern hardware. Thus, a database system must relax requirements such as pre-loading data and using full and precise auxiliary structures and take full advantage of the underlying hardware by being aware of the system architecture. By allowing partial and approximate data access, database systems can automatically trade storage space and precision for performance and vice versa.

Thesis statement

Traditional query processing relies on static assumptions about workload, dataset, and storage characteristics, thereby requiring applications to invest significant time and space pre-processing data. Building data access paths adaptively based on data and workload characteristics, as well as accuracy requirements, improves query processing performance and efficiency.

1.3.1 Towards Adaptive Data Access Methods

As data-centric applications become more complex, users face new challenges when exploring data, which are magnified with the ever-increasing data volumes. Database systems must embrace adaptivity and provide adaptive query processing approaches to enable efficient execution of shifting workloads. Data access methods have to dynamically adapt to evolving workloads and take advantage of relaxed accuracy requirements. Furthermore, query processing systems must be knowledgeable of available resources and maximize resource utilization thereby reducing waste. To this end we make the following contributions:

1. **Adaptive indexing for in-situ query processing** To achieve efficient data access despite dynamic workloads, we utilize state-of-the-art in-situ query processing techniques to minimize data-to-query time. Furthermore, we introduce a fine-grained logical partitioning scheme and combine it with a lightweight indexing strategy to provide near-optimal raw data access with minimal overhead in terms of execution time and memory footprint. To reduce the index selection overhead we propose an adaptive technique for on-the-fly partition and index selection using an online randomized algorithm.

2. **Adaptive data exploration using approximate access paths** Data scientists tolerate imprecise answers for better query performance [39]. We take advantage of the relaxed precision requirements to enable scaling of access paths despite ever-increasing datasets. Existing approaches [6, 70], either require full a priori knowledge of the workload to generate the required approximate data structures or improve performance through minimizing data access at query time. We design and demonstrate an adaptive approach which generates synopses as by-product of query execution and re-uses them for subsequent queries. It dynamically decides upon synopsis materialization and maintenance while being robust to workload and storage budget changes. To support interactive query performance for ever increasing datasets and dynamic exploratory workloads requires relaxing precision guarantees which enable the usage of approximate data structures and reduces the size of stored and processed data.

These aforementioned contributions serve as a platform to show the following key insights:

1. Taking advantage of data characteristics in files can complement in-situ query processing approaches by building data distribution-conscious access paths. Data properties such as ordering or clustering enable the construction of access paths spanning parts of a dataset reducing the cost of tuning and storage while minimizing data access costs and further reducing the data-to-insight time.

2. Ever-increasing datasets make precise access paths very expensive to build and store. Similarly, using data synopses as a drop-in replacement for indexes limits their benefits. On the contrary, integrating synopses as a first-class citizen in query optimization and by materializing synopses during query execution and re-using them across queries improves scalability and reduces pre-processing.

3. Static tuning decisions can be suboptimal in the presence of shifting exploratory workloads. Adapting access paths online, according to the workload while adhering to accuracy requirements is key to provide high query performance in the presence of workload changes.

1.3.2 Thesis Roadmap

This thesis is organized as follows. Chapter 2 provides the necessary background on concepts we utilize and extend in the context of this thesis. Chapter 3 shows how data access paths can reduce the cost of data access and reduce the storage overhead by adapting to data distribution and workload. Subsequently, Chapter 4 discusses how to speed-up query processing in distributed systems by reducing result precision, with no preprocessing time, and how to adapt to storage budget. Finally, Chapter 5 concludes the thesis.

2 Background and Related Work

In this thesis we discuss the design of database management system components to support exploratory query workloads. The large data volume and the lack of query predictability in exploratory workloads has major impact on data storage and access, as well as on database physical design. There is a plethora of background and related work regarding aspects which are central to this thesis. Specifically, we first discuss how a database system stores and accesses data and we motivate query processing over raw data files. Subsequently, we discuss techniques related to database tuning and indexing. Finally, we discuss approximate query processing techniques through sampling and sketching.

2.1 Data Access and Query Execution

Traditionally, database management systems (DBMS) designs assume data is stored and accessed using a unique data format. However, the growing variety in data formats along with the ever-growing data volume question the original paradigm creating alternative data access approaches. Many research efforts re-design the traditional data management architecture to address the challenges and opportunities associated with dynamic workloads and interactive data access. In this section we briefly present the various alternatives.

2.1.1 Loading and Querying data

To enable querying a dataset, a database management systems requires initially loading the dataset. Loading is a well-defined process describing the parsing and transformation of data from its original data format into a proprietary format used by the DBMS. Loading is an expensive process requiring time and storage space. Specifically, loading takes up a large fraction of overall workload execution time in both the DBMS and Hadoop ecosystems [60]. Furthermore, when loading a dataset, a DBMS is creating a copy of the original data in the DBMS-specific format, often requiring more space than the original dataset. Once the data is loaded, the DBMS is able to execute queries and build auxiliary structures (e.g., indexes) to

speed-up query execution. The internal operators of the DBMS are specifically implemented to process efficiently only the DBMS-specific format, thereby are dependent on the loading process.

2.1.2 Query processing over raw data files

The ever increasing variety and volume of data in many systems led to removing completely loading and adapting a novel data access paradigm named *in-situ*. This approach, treats any data format as a first-class citizen and internal database operators either use all data formats as native or perform *ad hoc* transformations in a pipelined fashion.

In-situ systems. The first system introducing the in-situ paradigm into traditional databases is NoDB, which treats raw data files as native storage of the DBMS, and introduces auxiliary data structures (positional maps and caches) to reduce the expensive parsing and tokenization costs of raw data access [10]. Extending NoDB, ViDa introduces code-generated access paths and data pipeline to adapt the query engine to the underlying data formats, layouts, and to the incoming queries [73–75]. Data Vaults [65, 71] and SDS/Q [22] perform analysis over scientific array-based file formats. SCANRAW [34] uses parallelism to mask the increased CPU processing costs associated with raw data accesses during in-situ data processing.

Scale-out raw data access. Hadoop-based systems such as Hive [110] can access raw data stored in HDFS. While such frameworks internally translate queries to MapReduce jobs, other systems follow a more traditional MPP architecture to offer SQL-on-Hadoop functionality [78, 87]. Hybrid approaches such as invisible loading [3] and Polybase [41] propose co-existence of a DBMS and a Hadoop cluster, transferring data between the two when needed.

This Thesis: Enabling data filtering over in-situ systems. The work presented in this thesis builds upon the in-situ paradigm and extends the design space. Specifically, in-situ DBMS approaches either rely on accessing the data via full table scans or require a priori workload knowledge and enough idle time to create the proper access paths. Our work, presented in Chapter 3, augments in-situ approaches systems by enabling data skipping and indexed accesses while constantly adapting its indexing and partitioning schemes to queries.

2.2 Physical Database Design

The performance of a database management system depends highly on its ability to minimize data access. To achieve that, along the years database systems utilize a variety of different constructs that reduce the amount of data a query has to access. We name such constructs physical design structures. Multiple types of structures exist, such as indexes, partitions, and materialized views. Despite reducing the amount of data access and speeding up queries, these structures are expensive to build, store and maintain. In Section 2.2.1 we describe key

structures and give details considering the cost of building and storing a physical design structure.

The advent of dynamic workloads and the increasing volume of datasets increases the cost of maintaining numerous physical design structures. To reduce this cost and to automatize the process, DBMS vendors created automated tools enabling the construction and destruction of such constructs as well as advisors suggesting which physical design structures should a DBMS administrator construct. In Section 2.2.2 we give more details considering the challenges in designing an automated physical design tuner.

2.2.1 Physical design structures

Every physical design structure improves the performance of a set of queries, and requires pre-processing time and additional storage space. We describe three structures that are central to this thesis along with key state-of-the-art implementations.

Indexes

A database index is a data structure that improves the speed of data retrieval. Indexes remove the requirement for full scan searches and help quickly locate data. There is a vast collection of index structures with different capabilities, performance, and initialization/maintenance overheads [18, 19]. In the context of this thesis we sub-divide index structures into two categories (i) value-position and (ii) value-existence indexes, that offer good indexing for point and range queries. Value-position indexes map a value to its location in the file and include the B^+ tree and Hash indexes and their variations [17]. Common value-existence indexes are Bloom filters [23], Bitmap indexes [20, 92, 106], and zone maps [88]. They are lightweight and can provide the information whether a value is present in a given dataset. Value-existence indexes are frequently used in scientific workloads [35, 108, 114]. For scale-out systems, SQL Server PDW [49] and AsterixDB [11] propose indexes for data stored in HDFS and for external data in general. Both approaches use similar techniques to single-node indexing while addressing the problem of distributed datasets.

Materialized views

To speed-up query processing and reduce data access for predictable workloads, DBMS pre-compute a set of results which are re-usable across many queries. These results are called materialized views. Materialized view management involves three major problems: (i) selecting and generating views, (ii) updates and (iii) utilization in a query.

Selecting and building views. Deciding on appropriate materialized views for a workload is a non-trivial process. Materialized views require considerable space to be stored, thereby given a storage budget, there is a limited number of materialized views that can be generated. The

decision-making algorithm must trade storage for speed-up. The goal is to maximize speed-up while adhering to the pre-defined storage budget. This process requires prior knowledge of the workload (predicates, joins, query frequency etc.) as well as data distribution (to calculate prospective view size).

Updating views. Materialized view maintenance requires considerable book-keeping. Update heavy workloads require constant updates of views as updates on the original tables have to be propagated to the views. Various techniques have been designed to address this issue [101].

View matching. Given a query and a set of materialized views, deciding whether any of the views can speed-up the query is also non-trivial. Determining if a query can be computed from a materialized view, the DBMS must ensure that (i) the materialized view must contain all rows that are required by the query and (ii) to choose the best available view for this specific query. Furthermore, utilizing view should speed-up queries thus, *view matching* has to be an efficient process offering considerable speed-up.

Database Partitions

Given a relational table, it can be physically subdivided into smaller disjoint sets of tuples (partitions), allowing tables to be stored, managed and accessed at a finer level of granularity [82]. Deciding on partitions is non-trivial and depends highly on the data distribution and query workload.

Assuming a predictable workload, offline partitioning approaches [8, 54, 94, 118] present physical design tools that automatically select the proper partition configuration for a given workload to improve performance.

On the other hand, when the workload is either unknown or unpredictable, online partitioning [68] monitors and periodically adapts the database partitions to fit the observed workload. Furtado et al. [47] combine physical and virtual partitioning to fragment and dynamically tune partition sizes for flexibility in intra-query parallelism. Shinobi [113] clusters hot data in horizontal partitions which it then indexes, while Sun et al. [109] use a bottom-up clustering framework to offer an approximate solution for the partition identification problem.

2.2.2 Automated Tuners

Database physical design is of paramount importance for database vendors. However, the correct selection of structures which provide high performance for a system depends highly on the database system administrator (DBA). In order to support the DBA decisions, database vendors have developed automated tuners which either provide suggestions or make automatic decisions on index construction.

Such tuners can be essentially divided into two categories depending on the decision making

approach (i) offline tuners and (ii) online tuners. A third alternative to these two options are physical design structures which automatically tune themselves when the workload is evolving. We call this approach "adaptive tuning".

Offline tuning. A typical offline physical designer tries to solve the following problem: Given a workload W and a set of constraints C (e.g. storage budget), find a configuration P of physical structures which satisfies the constraints C and minimizes the execution cost for W. Most commercial DBMS vendors offer physical designers in their products (e.g., SQL Server Tuning Advisor [32], DB2 Design Advisor [111], Oracle SQL Access Advisor [40, 42]). A database administrator typically examines the output of the physical design process, verifies the usefulness of the proposed configuration and decides what structures to create inside the database. A physical tuner relies on the database query optimizer to decide on the usefulness of a given structure. This process also known as "what-if" analysis, simulates the presence of different structures and decides whether their utilization would be beneficial. Such an approach guarantees the correctness of the decision given the utilization of the same optimizer when executing the queries (different cost models may result to different decisions).

Online tuning. Offline physical designer assumes that the workload is static and the set of queries within workload W is a good representation of the future. However, in modern exploratory workloads where queries are data-driven, static decisions may be sub-optimal. To address such cases, online tuners re-evaluate physical design decisions. COLT [104] continuously monitors the workload and periodically creates new indexes and/or drops unused ones. COLT adds overhead on query execution because it obtains cost estimations from the optimizer at runtime. A "lighter" approach requiring fewer calls to the optimizer has also been proposed [26]. Similarly, [7] periodically re-evaluates the physical design decision on a window at-a-time basis assuming that the past window is a good representation of the upcoming queries.

Adaptive Indexing. In order to avoid the full cost of indexing before workload execution, Adaptive Indexing incrementally refines indexes during query processing. In the context of in-memory column-stores, Database Cracking approaches [51, 61–63, 100] create a duplicate of the indexed column and incrementally sort the data it according to the incoming workload, thus reducing memory access. HAIL proposes an adaptive indexing approach for MapReduce systems [102]. ARF is an adaptive value-existence index similar to Bloom filters, yet useful for range queries [12].

This thesis: Adaptive database tuning over raw data files. Chapter 3 of this thesis is motivated by the high cost of physical design tuning and takes advantage of insights from online tuning and adaptive indexing. Considering physical design structures, our approach Slalom uses a combination of value-position and value-existence indexes and improves system scalability by reducing the size of the indexes. Regarding physical re-organization, Slalom presents a non-intrusive, flexible partitioning scheme that creates logical horizontal partitions by exploit-

ing data skew. Additionally, Slalom continuously refines its partitions during query processing without requiring a priori workload knowledge. Similarly to online tuning, our approach also focuses on the problem of selecting an effective set of indexes but Slalom builds indexes on a per-partition granularity. Slalom also populates indexes during query execution in a pipelined fashion instead of triggering a standalone index building phase. Slalom aims to minimize the cost of index construction decisions and the complexity of the costing algorithm. Finally, similar to adaptive indexing, Slalom does not index data upfront and builds indexes during query processing and continuously adapts to the workload characteristics. However, contrary to adaptive indexing that duplicates the whole indexed attribute upfront, Slalom's gradual index building allows its indexes to have small memory footprint by indexing both the targeted value ranges, and the targeted attributes.

2.3 Approximate Query Processing

There is a large collection of recent work on approximate query processing. In this section we initially describe data summaries also known as synopses, and subsequently we discuss the different ways in which approximation is integrated into query execution.

2.3.1 Types of synopses

We denote as synopsis a construct summarizing information of a full dataset and is substantively smaller than the base dataset. Specifically, synopses include samples as well as data structures such as bloom filters, sketches or histograms.

In this section we will briefly describe samples, bloom filters and sketches. All satisfy the following requirements, which are imperative for high performance: (a) they are partitionable, i.e., they can be constructed over massively parallel platforms (e.g., Spark) and (b) they are pipelineable, such that they can be built with a single pass over the data. These synopses are utilized in our Slalom and Taster systems which are introduced in Chapter 3 and Chapter 4. For a more in depth analysis of other synopses, we refer the reader to Cormode et.al. [37].

Samples. A sample is a subset of the original dataset. In the bibliography there is a large variety of different types of samples depending on the approach used to extract the subset. In the context of this work we describe in depth two types of samplers – uniform and distinct. We describe the samples based on its extraction process from the original dataset and the "sampler" operator that performs that process. A sampler scans all input rows, and lets only a subset of them to pass through. To enable scaling of query results from using a sample to the ones of the full dataset, each sampler appends an additional attribute that represents the weight associated with the row. For example, given a query calculating the SUM of a column, for every tuple of the sample with value t_i and weight w_i, thus a system will return $t_i \times w_i$.

Uniform sampler. The uniform sampler Γ^{U}_{p} samples without replacement, letting a row pass

through with probability p at random. For every row, the weight is set to $1/p$. This sampler is both pipelineable and partitionable, and its memory footprint during construction is approximately equal to the memory footprint of the desired sample size. Alternative uniform sampling implementations exist, which however require multiple passes over the data [37] and are therefore not pipelinable.

Distinct sampler. Even though the uniform sampler has low execution overhead, it does not have good statistical properties in more complex workloads, e.g., in queries containing group-by it may miss whole groups, whereas in join queries, it may miss an arbitrary large number of join keys. Prior works cope with such cases by generating stratified samples. Stratified sampling guarantees the existence of all groups for specific attributes (stratification attributes) and a minimum number of tuples per group. However, stratified sampling operators are blocking operators and require two passes over the data. This second pass of data increases execution overhead of such an operator and blocks execution. Alternatively, the distinct sampler [37,69,70], guarantees that at *least a certain number* of rows pass per distinct combination of values of a column set.

Distinct sampler works as follows: given a set of stratification attributes $\mathscr{A}$, a number δ, and probability p, the distinct sampler $\Gamma^{\mathsf{D}}_{p,\mathscr{A},\delta}$ passes at least δ rows for every distinct combination of values of the columns in $\mathscr{A}$. Subsequent rows with the same value are let through with probability p, uniformly-at-random. The weight of each row is set correspondingly: If the row passes because of the frequency check, its weight is set to 1, whereas if it passes due to the probability check, its weight is set to $1/p$. In terms of implementation, distinct sampling is implemented efficiently by using a heavy-hitters sketch that requires space logarithmic to the number of rows [37].

Distinct sampler is pipelinable by design, as it requires only a single pass over the data. To make it partitionable, given the sampler operator distribution factor D (the number of operator instances), we adjust the minimum number of rows required from each operator instance from δ to $\delta + D\epsilon$ with ϵ being a variable addressing variations in data distribution. As per [70], ϵ is set to δ/D, which builds on the assumption that data is distributed uniformly across instances.

Sketches. Sketch is a probabilistic data structure that stores a specific piece of information over a dataset. For example, count-min sketches (CM-sketch) store frequencies of specific events [38]. We briefly summarize these here, to the extent required for understanding the work in Chapter 4. A count-min sketch consists of a 2-dimensional array ($w \times d$) of counters (integers), accompanied with d pairwise independent hash functions that uniformly map each item from the domain space (each potential key) to one counter per array row. Let counter i at row j be denoted as $A[i][j]$, and hash function corresponding to row j be denoted as h_j. Adding an item x to a sketch is achieved by finding the corresponding counters from the sketch, i.e., $A[h_j(x)][j]$ for $j = \{1,\dots d\}$, and increasing these by one (or by the frequency of x). The sketch has a memory footprint of a few MB and can be constructed on-the-fly during

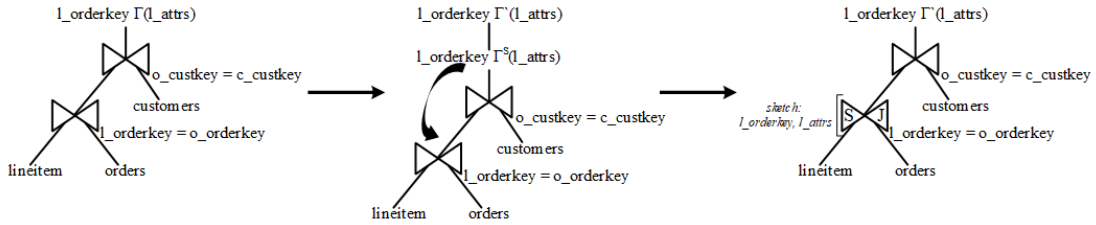

Figure 2.1 – TPC-H Q3 plan transformation using Sketch-Join.

query answering. After construction, the sketch is used as an *approximate key-value store* for estimating the frequency $\widehat{f}(x)$ of any item x, as follows: $\widehat{f}(x) = \min\left(A[h_j(x)][j] | j = \{1,\ldots d\}\right)$. When $d = O(1/\delta)$ and the number of columns of the sketch is set to $O(1/\epsilon)$, the estimate is within range ϵN of the real answer, with probability at least $1-\delta$ (variable N represents the L1 norm of the frequencies). Construction of count-min sketches is fully partitionable. Therefore, each node in the cluster builds sketches for its own data, and all sketches for one dataset are added pair-wise to get a sketch representing the whole RDD.

Sketches have very small storage footprint have constant access and update time however they can answer only specific queries. Such queries are, e.g., nested queries containing EXISTs which can be approximated with Bloom filters [23], distinct counts and join size estimations with Bloom filters [95], FM-sketches [46], and AMS-sketches [13].

Sketch-join. Besides simple aggregations, the sketch also supports aggregations over joins. The Sketch-Join operator builds a sketch on the relation over which the aggregation takes place and uses as key the join key and as a value the executed aggregation for the tuple. This sketch is subsequently used in a similar fashion as a hash index in the hash-join algorithm – the Sketch-Join operator probes the sketch and returns the appropriate aggregate. Two operations are combined in Sketch-Join: partial aggregation (i.e., the computation of aggregation over subsets of the data), and summarization of the aggregates. The computation of partial aggregates reduces the memory footprint of the Join operator, which now has to store less columns (as it returns only the aggregation result), and the sketch is much smaller than the hash index. The reduced size of the sketch (a few MB as opposed to possibly several GB for a sample of a large table, or a hash index) makes sketches ideal for materialization and re-use in subsequent queries.

As an example, Figure 2.1 (left) presents the plan for `Q3` of industrial benchmark TPC-H. The plan includes an aggregation over a large set of attributes from relation `lineitem` and the grouping is done over the `l_orderkey` attribute which is also the join key between relations `lineitem` and `orders`. [1] For this query, a Sketch-Join (Figure 2.1 right) over sampling-based estimates of `lineitem` or `orders` can be utilized instead of executing a hash join, since sketches are more compact than hash tables.

[1] We denote the aggregation attribute set `l_attributes`.

Bloom filter. A Bloom filter is a space-efficient probabilistic data structure, used to test whether an element is a member of a set. The approximation in Bloom filters comes from the possible existence of false positive matches. On the other hand, a Bloom filter has no false negatives. False positives denote the case when a Bloom filter will respond that a value exists in a set while it does not and False negatives denote the case when a Bloom filter will respond that a value does not exists while it does. The original Bloom filter design [23], enables the insertions of elements however elements cannot be removed. Later designs such as Counting Filters [44], address this issue. The popularity of Bloom filters stems from their small size, their small false positive error and their constant and efficient response time.

2.3.2 Approximation in Query Execution

The synopses described in the previous section can be utilized in numerous ways to speed-up query execution. In this section we separate the approaches into three categories: (i) *offline approximation*, (ii) *online approximation*, and (iii) *online aggregation*.

Offline approximation. Traditional offline approximate query processing has two stages the (i) offline and (ii) online stage. During the offline stage, the user decides on a representative set of queries which corresponds to the queries that will be issued later. Based on those queries he creates a set of samples of the original dataset. Subsequently, during the online phase, a submitted query is matched to the most appropriate pre-computed sample and executed upon it.

Offline approximation requires some degree of knowledge over the upcoming queries. The different approaches in the bibliography differ primarily on the nature of samples that they maintain and the technique to choose the set of samples to generate. Congressional sampling, STRAT and BlinkDB [4, 6, 30] provide algorithms to compute the best set of uniform and stratified samples, subject to a storage budget. In the same line, other works maintain additional data structures to better support skewed datasets and to reduce the size of samples [21, 29, 107]. AQUA [5] and VerdictDB [96] instead act as a middleware between users and traditional database systems, by rewriting user queries to take advantage of precomputed samples. VerdictDB is particularly interesting as it proposes a novel error estimation technique called variational subsampling, which enables smaller samples. Similarly, Sample+Seek [43] introduces measure-biased sampling which takes advantage of indexes to create more efficient samples and provide error guarantees for GROUP BY queries with many groups. AQP++ [98] blends AQP with aggregate precomputation, such as data cubes, to handle aggregate relational queries. Such a unified approach balances preprocessing time and query runtime.

Online Approximation. To address the limitations that source from the uncertainty of the future query workload, online approximation techniques remove the need for workload knowledge and approximate queries at runtime. Quickr [70] follows an online sampling approach, where samples are taken during query execution. In particular, samplers are

injected into the query plan to reduce network and computation load. However, Quickr performance gains are constrained by the I/O cost since the system still needs to read the full input for every query. Similarly, Galakatos et al. [48] build and re-use samples for a data exploration scenario.

Online aggregation. Similarly, to online approximation, online aggregation requires no pre-processing and approximation is introduced at query runtime. Specifically, (OLA) [58, 67, 93, 115] instead of sampling over data, they estimate the answer by looking at progressively increasing portions of the data, until a user determines that the answer quality is sufficient. EARL [80] and ABS [117] use bootstrapping to produce multiple estimators from the same sample. Finally, iOLAP [116] models online aggregation as incremental view maintenance with uncertainty propagation.

This thesis: Adapting approximation to workload. The approach presented in Chapter 4 of this thesis is motivated by the high storage cost and static decisions of offline approximation and the relevantly low speed-up of online approximation.

Offline approximation approaches are designed based on static assumptions about future queries. Thus, they require workload knowledge, and they undergo a time-consuming pre-processing operation for sample preparation. Both are limiting properties of these methods, since in modern data analytics setups (e.g., data exploration), the analyst typically starts with little knowledge about data. Hence, she can hardly predict the future workload, or the time that she will spend analyzing the new data, in order to decide whether an extensive sampling preparation will be beneficial. Furthermore, these methods fail when the actual queries diverge from the predicted workload that was used for constructing the synopses. Our system Taster, presented in Chapter 4, efficiently addresses these constraints since it constructs and adapts the synopses online, during query execution. Still, it can also capitalize on user hints – when there is such a possibility – to construct some samples in an offline phase, using more advanced sampling strategies. For example, in Chapter 4 we show how to incorporate a sampling scheme from VerdictDB. Techniques presented in Sample+Seek [43] and AQP++ [98] are also prime candidates for integration into Taster, to further speed-up query execution by taking advantage of state-of-the-art sampling and precomputed aggregates.

Considering online approximation, existing approaches, assume that the user builds queries incrementally, allowing the system to generate samples while the user is further expanding his query (e.g., adding a filter). Furthermore, in existing approaches samples are built only over base relations, not taking advantage of intermediary results. In our work, Taster extends the online approximation techniques of Quickr in several non-trivial ways. First, it materializes/stores some of these synopses for re-use in future queries. The decision as to which synopses should be stored relies on a formal model, which enables adaptivity to the workload and to the shifting user interests, and is amenable to efficient approximations. Second, it incorporates additional types of synopses, beyond samples. Finally, it incorporates hints for offline synopsis construction, thereby exhibiting the best properties of both online and offline AQP.

3 Adaptive in-situ Partitioning and Indexing

The constant flux of data and queries alike has been pushing the boundaries of data analysis systems. The increasing size of raw data files has made data loading an expensive operation that delays the data-to-insight time. To alleviate the loading cost, in-situ query processing systems operate directly over raw data and offer instant access to data. At the same time, analytical workloads have increasing number of queries. Typically, each query focuses on a constantly shifting – yet small – range. As a result, minimizing the workload latency, requires the benefits of indexing in in-situ query processing.

This chapter, presents an online partitioning and indexing scheme, along with a partitioning and indexing tuner tailored for in-situ querying engines. The proposed system design improves query execution time by taking into account user query patterns, to (i) partition raw data files *logically* and (ii) build lightweight *partition-specific* indexes for each partition.

We build an in-situ query engine called Slalom to showcase the impact of our design. Slalom employs adaptive partitioning and builds non-obtrusive indexes in different partitions on-the-fly based on light-weight query access pattern monitoring. As a result of its light-weight nature, Slalom achieves efficient query processing over raw data with minimal memory consumption. Our experimentation with both micro-benchmarks and real-life workloads shows that Slalom outperforms state-of-the-art in-situ engines, and achieves comparable query response times with fully indexed DBMS, offering lower cumulative query execution times for query workloads with increasing size and unpredictable access patterns.

3.1 Introduction

Data-intensive applications in various domains generate and collect massive amounts of data at a rapid pace. New research fields and applications (e.g., network monitoring, sensor data management, clinical studies, etc.) emerge and require broader data analysis functionality to rapidly gain deeper insights from the available data. In practice, analyzing such datasets become costlier as data sizes grow.

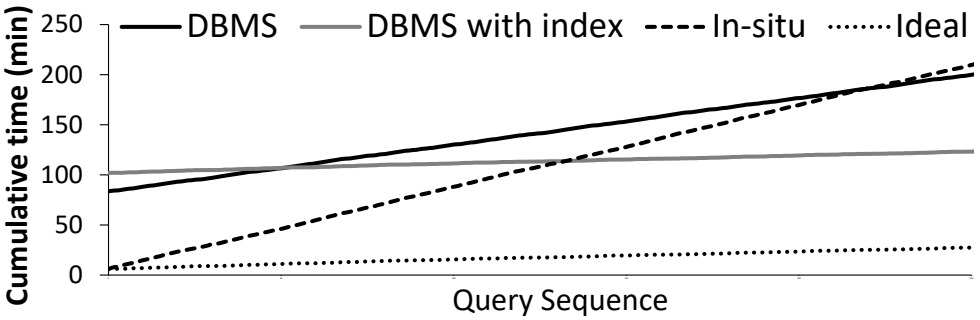

Figure 3.1 – Comparing cumulative execution time of traditional DBMS, In-situ DBMS with the ideal case.

Big Data, Small Queries. The trend of exponential data growth due to intense data generation and data collection is expected to persist. However, recent studies of the data analysis workloads show that typically only a small subset of the data is relevant and ultimately used by analytical and/or exploratory workloads [2, 33]. In addition, modern businesses and scientific applications require interactive data access, which is characterized by *no or little a priori workload knowledge* and constant *workload shifting* both in terms of projected attributes and selected ranges of the dataset.

The Cost of Loading, Indexing, and Tuning. Traditional data management systems (DBMS) require the costly steps of *data loading*, *physical design*, and *index building* in order to offer interactive access over large datasets. Given the size of the data involved, any transformation, copying, and preparation steps over the data introduces substantial delays before the data can be utilized, queried, and provide useful insights [3, 10, 65]. The lack of *a priori* knowledge of the workload makes the physical design decisions impossible because cost-based advisors rely heavily on past or sample workload knowledge [8, 31, 45, 56, 118]. Exploratory workloads often exhibit sudden workload shifts that depend on the observed data values and on results of the ongoing analysis. Hence, investments on indexing and physical design using current workload information can be nullified unexpectedly.

Querying Raw Data Files is not Enough. Recent efforts opt to directly query raw files [3, 10, 22, 34, 60, 75] to reduce the data-to-insight cost. These *in-situ* systems avoid the costly initial data loading step, and allow the execution of declarative queries over external files without duplicating or "locking" data in a proprietary database format. Further, they concentrate on reducing costs associated with raw data accesses (e.g., parsing and converting data fields) [10, 34, 75]. Finally, although recent scientific data management approaches index raw data files using file-embedded indexes, they do it in a workload-oblivious manner, or requiring full *a priori* workload knowledge [22, 114]. Hence, they still have to pay the upfront cost of full index building, missing the opportunity to make on-demand physical design decisions during workload execution, which helps to fully benefit from avoiding data loading.

Figure 3.1 visualizes the benefits of state-of-the-art in-situ query processing when compared with a full DBMS, as well as, what the ideal in-situ query performance should be (dotted line). After the unavoidable first table scan, ideally, in-situ queries need to access only data relevant to the currently executed query. The y-axis shows the cumulative query latency, for an increasing number of queries with fixed selectivity on the x-axis. By avoiding the costly data loading phase the in-situ query execution system (dashed line) can start answering queries very quickly. On the other hand, when a DBMS makes an additional investment on full indexing (solid grey line), the data-to-query latency initially increases; however, later it pays off as the number of queries issued over the same (raw) dataset increases. Eventually, the cumulative query latency for an in-situ approach becomes larger than the latency of a DBMS equipped with indexing. When operating over raw data, *ideally*, we want after the initial – unavoidable – table scan, to collect enough metadata to allow future queries to access only the useful portion of the dataset.

Adaptive Partitioning and Fine-Grained Indexing. We use the first table scan to generate partitioning and lightweight indexing hints which are further refined by the data accesses of (only a few) subconsequent queries. During this refinement process, the dataset is partially indexed dynamically adapting to three key workload characteristics: (i) data distribution, (ii) query type (e.g., point query, range query), and (iii) projected attributes. Workload shifts lead to varying (a) selected value ranges, (b) query selectivity, (c) dataset areas are relevant for a query, and (d) projected attributes.

This paper proposes an online partitioning and indexing tuner for in-situ query processing which when plugged into a raw data query engine, offers *fast queries over raw data files*. The tuner reduces data access cost by: (i) *logically partitioning* a raw dataset to virtually break it into smaller manageable chunks *without physical restructuring*, and (ii) choosing *appropriate indexing strategies over each logical partition* to provide efficient data access. The tuner dynamically adapts the partitioning and indexing scheme as by-product of query execution. It continuously collects information regarding the values and access frequency of queried attributes at runtime. Based on this information, it uses a randomized online algorithm to define the logical partitions. For each logical partition, the tuner estimates the cost-benefit of building partition-local index structures considering both approximate membership indexing (i.e., Bloom filters and zone maps) and full indexing (i.e., bitmaps and B^+ trees). By allowing fine-grained indexing decisions our proposal makes the decision of the index shape at the level of each partition rather than the overall relation. This has two positive side-effects. First, there is no costly investment for indexing that might prove unnecessary. Second, any indexing effort is tailored to the needs of data accesses on the corresponding range of the dataset.

Efficient In-Situ Query Processing with *Slalom*. We integrate our online partitioning and indexing tuner to an in-situ query processing prototype system, *Slalom*, which combines the tuner with a state-of-the-art raw data query executor. Slalom is further augmented with index structures and uses the tuner to decide how to partition and which index or indexes to build for

each partition. In particular, Slalom logically splits raw data into partitions and selects which fine-grained index to build, per-partition based on how "hot" (i.e., frequently accessed) each partition is, and what types of queries target each partition. Moreover, Slalom populates binary caches (of data converted from raw to binary) to further boost performance. Slalom adapts to workload shifts by adjusting the current partitioning and indexing scheme using a randomized cost-based decision algorithm. Overall, the logical partitions and the indexes that Slalom builds over each partition provide performance enhancements without requiring expensive full data indexing or data file re-organization, all while adapting to workload changes.

Contributions. The contributions presented in this chapter are the following:

- We present a logical partitioning scheme of raw data files that enables fine-grained indexing decisions at the level of each partition. As a result, light-weight per-partition indexing provides near-optimal data access.
- The light-weight partitioning allows our approach to maintain the benefits of in-situ approaches. In addition, the granular way of indexing (i) brings the benefit of indexing to in-situ query processing, (ii) having low index building cost, and (iii) small memory footprint. These benefits are highlighted as the partitioning and indexing decisions are refined on-the-fly using an online randomized algorithm.
- We enable in-place and append-like updates. We exploit specialized hardware (GPUs and CRC checksum units) to reduce the raw data file monitoring cost recognize incoming updates, and find the smallest possible changes needed to the partitioning and indexing strategies employed. The discovery of the minimal changeset reduces the index correction overhead and improves query execution performance in the presence of updates.
- We integrate our partitioning and indexing tuner into our prototype state-of-the-art in-situ query engine *Slalom*. We use synthetic and real-life workloads to compare the query latency of (i) Slalom, (ii) a traditional DBMS, (iii) a state-of-the-art in-situ query processing engine, and (iv) adaptive indexing (cracking). Our experiments show that, even when excluding the data loading cost, Slalom offers the fastest cumulative query latency. In particular, Slalom outperforms (a) state-of-the-art disk-based approaches by one order of magnitude, (b) state-of-the-art in-memory approaches by 3.7× (with 2.45× smaller memory footprint), and (c) adaptive indexing by 19% (having 1.93× smaller memory footprint). Finally, we examine the performance of Slalom in presence of both in-place and append-line updates.

To our knowledge, Slalom is the first system that proposes the use of a randomized online algorithm to select which workload-tailored, index structures should be built per partition of the data file. This approach offers constant, and minimal, decision time.

Outline. The remainder of this chapter is organized as follows: Section 3.2 presents the architecture of Slalom and gives an overview of its design. Section 3.3 presents the online

tuner and describes its partitioning and indexing cost models. Section 3.3.3 presents the techniques enabling efficient data updates for in-situ query processing. We experimentally demonstrate the benefits of Slalom in Section 3.4 and we conclude in Section 3.5.

3.2 The SLALOM System

Slalom uses adaptive partitioning and indexing to provide inexpensive index support for *in-situ* query processing while adapting to workload changes. Slalom accelerates query processing by skipping data and minimizes data access cost when this access is unavoidable. At the same time, it operates directly on the original data files without need for physical restructuring (i.e., copying, sorting).

Slalom incorporates state-of-the-art in-situ querying techniques and enhances them with logical partitioning *with no physical data movement* and fine-grained indexing, thereby reducing the amounts of accessed data. To remain effective despite workload shifts, Slalom introduces an online partitioning and indexing tuner, which calibrates and refines logical partitions and secondary indexes based on data and query statistics. Slalom treats data files as relational tables to facilitate the processing of read-only and append-like workloads. The rest of this section focuses on the architecture and implementation of Slalom.

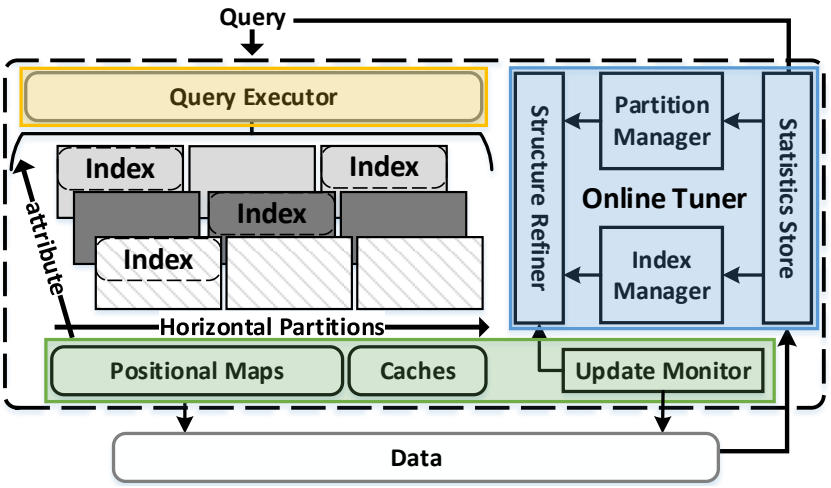

Figure 3.2 – The architecture of Slalom.

3.2.1 Architecture Overview

Figure 3.2 presents the architecture of Slalom. Slalom combines an online partitioning and indexing tuner with a query executor featuring in-situ querying techniques. The core components of the tuner are the *Partition Manager*, which is responsible for creating logical partitions over the data files, and the *Index Manager*, which is responsible for creating and maintaining indexes over partitions. The tuner collects statistics regarding the data and query

access patterns and stores them in the *Statistics Store*. Based on those statistics, the *Structure Refiner* evaluates the potential benefits of alternative configurations of partitions and indexes. Furthermore, Slalom uses *in-situ* querying techniques to access data. Specifically, Slalom uses auxiliary structures (i.e., positional maps and caches) which minimize raw data access cost. During query processing, the *Query Executor* utilizes the available data access paths and orchestrates the execution of the other components. Finally, the *Update Monitor* examines whether a file has been modified and adjusts the data structures of Slalom accordingly.

Slalom Scope. The techniques of Slalom are applicable to any tabular dataset. Specifically, the scan operator of Slalom uses a different specialized parser for each underlying data format. This work concentrates on queries over delimiter-separated textual CSV files, because CSV is the most popular structured textual file format. Still, the yellow- and blue-coded components of Figure 3.2 are applicable over binary files, which are the typical backend of databases and are also frequently used in scientific applications (e.g., high-energy physics, DNA sequencing, GIS). We discuss further Slalom's extensibility in Section 3.2.4.

Reducing Data Access Cost. Slalom launches queries directly over the original raw data files, without altering or duplicating the files by ingesting them in a DBMS. That way, Slalom avoids the initialization cost induced by loading and offers instant data access. Similarly to state-of-the-art in-situ query processing approaches [10, 34] Slalom mitigates the overheads of parsing and tokenizing textual data with positional maps [10] and partial data caching.

Positional maps are populated on-the-fly and maintain *structural* information about an underlying textual file, that is, the positions of each attribute for each row. This information is used during query processing to "jump" to the exact position of an attribute or as close as possible to an attribute, significantly reducing the cost of tokenizing and parsing when a tuple is accessed. Furthermore, Slalom builds binary caches of fields that are already converted to binary to reduce parsing and data type conversion costs of future accesses.

Statistics Store. Slalom collects statistics during query execution and utilizes them to (i) detect workload shifts and (ii) enable the tuner to evaluate partitioning and index configurations. Table 3.1 summarizes the statistics about *Data* and *Queries* that Slalom gathers per data file. *Data statistics* are updated after every partitioning action and include the per-partition standard deviation (dev_i) of values, mean (m_i), max (max_i) and min (min_i) values. Additionally, Slalom keeps as global statistics the physical page size ($Size_{page}$) and file size ($Size_{file}$). Regarding *Query statistics*, Slalom maintains the number of queries since the last access (LA_i), the percentage of queries accessing each partition (access frequency AF_i), and the average query selectivity (sel_i). The full scan cost over a partition ($C_{i_{fullscan}}$) and the indexing cost for a partition ($C_{i_{build}}$) is calculated by considering the operator's data accesses. The aforementioned statistics are used in the cost formulas of Sections 3.3.1 and 3.3.2.

Partition Manager. The Partition Manager recognizes patterns in the dataset and logically

Data (partition i)	m_i	Mean value
	min_i	Min value
	max_i	Max value
	dev_i	Standard deviation
	DV_i	#distinct values
Data (global)	$Size_{page}$	Physical page size
	$Size_{file}$	File size
Queries (partition i)	$C_{i_{build}}$	Index building cost
	$C_{i_{fullscan}}$	Full scan cost
	LA_i	#queries since last access
	AF_i	Partition access frequency
	sel_i	Average selectivity (0.0-1.0)

Table 3.1 – Statistics collected by Slalom per data file during query processing and used to (i) decide which logical partitions to create, and (ii) select the appropriate matching indexes.

divides the file into contiguous non-overlapping chunks to enable fine-grained access and indexing. The Partition Manager specifies a logical partitioning scheme for each attribute in a relation. Each partition is internally represented by its starting and ending byte within the original file. The logical partitioning process starts the first time a query accesses an attribute. The Partition Manager triggers the Structure Refiner to iteratively fine-tune the partitioning scheme with every subsequent query. All partitions progressively reach a state in which there is no benefit from further partitioning. The efficiency of a partitioning scheme depends highly on the data distribution and the query workload. Therefore, the Partition Manager adjusts the partitioning scheme based on value cardinality.

Index Manager. The Index Manager estimates the benefit of an index over a partition and suggests the most promising combination of indexes for a given attribute/partition. For every new index configuration, the Index Manager invokes the Structure Refiner to build the selected indexes during the execution of the next query. Every index corresponds to a specific data partition. Depending on the access pattern of an attribute and the query selectivity, a single partition may have multiple indexes. Slalom chooses indexes from two categories based on their capabilities: (i) *value-existence* indexes, which respond whether a value exists in a dataset (e.g., Bloom filters), and (ii) *value-position* indexes, which return the positions of a value within the file (e.g., B^+ tree). The online nature of Slalom imposes a significant challenge not only on which indexes to choose but also on when and how to build them with low cost. The Index Manager monitors previous queries to decide which indexes to build and when to build them; timing is based on an online randomized algorithm which considers (i) statistics on the cost of full scan ($C_{i_{fullscan}}$), (ii) statistics on the cost of building an index ($C_{i_{build}}$), and (iii) partition access frequency (AF_i).

Update Monitor. The main focus of Slalom is read-only and append workloads. Still, to provide query result consistency, the Update Monitor checks the input files for both appends

and in-place updates at real-time. Slalom enables append-like updates without disturbing query execution by dynamically adapting its auxiliary data structures. Specifically, Slalom creates a partition at the end of the file to accommodate the new data, and builds binary caches, positional maps and indexes over them during the first post-update query. In-place updates require special care in terms of positional map and index maintenance because they can change the internal file structure. Slalom reacts to in-place updates during the first post-update query by identifying the updated partitions, updating the positional map, and recreating the other corresponding structures.

3.2.2 Implementation

We implement Slalom from scratch in C++. Slalom's query engine uses tuple-at-a-time execution based on the Volcano iterator model [52]. The rest of the components are implemented as modules of the query engine. Specifically, the *Partitioning* and *Indexing* managers as well as the *Structure Refiner* connect with the Query Executor. Furthermore, the *Statistics Store* runs as a daemon, gathering the data and query statistics and persisting them in a catalog.

Slalom reduces raw data access cost by using vectorized parsers, binary caches, and positional maps. The CSV parser uses SIMD instructions; it consecutively scans a vector of 256 bytes from the input file and applies a mask over it to identify delimiters. Slalom populates a positional map for each CSV file accessed. To reduce memory footprint, the positional map stores only delta distances for each tuple and field. Specifically, to denote the beginning of a tuple, the positional map stores the offset from the preceding tuple. Furthermore, for each field within a tuple, the positional map stores only the offset from the beginning of the tuple. The Partition Manager maintains a mapping between partitions and their corresponding positional map portions.

Slalom populates binary caches at a partition granularity. When a query accesses an attribute for the first time, Slalom consults the positional map to identify the attribute's position, and then caches the newly converted values. To improve insertion efficiency, Slalom stores the converted fields of each tuple as a group of columns. If Slalom opts to convert an additional field during a subsequent query, it appends the converted value to the current column group.

Slalom also populates secondary indexes at a partition granularity; for each attribute, the indexes store its position in the file and its position in the binary cache (when applicable). Slalom uses a cache friendly in-memory B^+ tree implementation. It uses nodes of 256 bytes that are kept 60% full. To minimize the size of inner nodes and make them fit in a processor cache line, the keys in the nodes are stored as deltas. Furthermore, to minimize tree depth, the B^+ tree stores all appearances of a single value in one record.

The Structure Refiner monitors the construction of all auxiliary structures and is responsible for memory management. Slalom works within a memory area of pre-defined size. The indexes, positional maps, and caches are placed in the memory area. However, maintaining

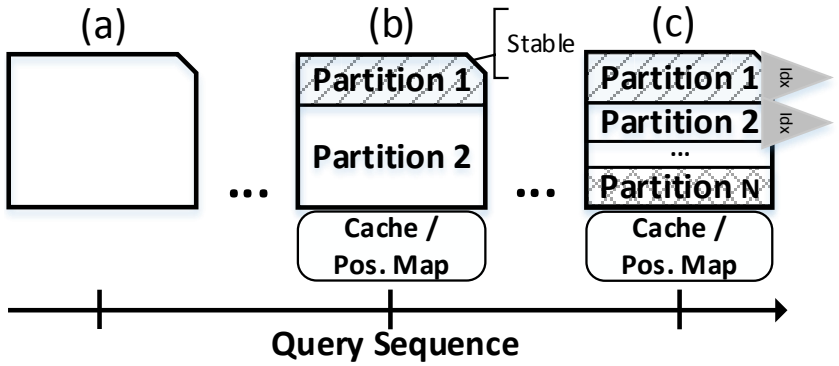

Figure 3.3 – Slalom execution.

caches of the entire file and all possible indexes is infeasible. Thus, the Structure Refiner dynamically decides, on a per-partition basis, which structure to drop so Slalom can operate when resources are limited.

3.2.3 Query Execution

Figure 3.3 presents an overview of a query sequence execution over a CSV file. During each query, Slalom analyzes its current state in combination with the workload statistics and updates its auxiliary structures. In the initial state (a), Slalom has no data or query workload information. The first query accesses the data file without any support from auxiliary structures; Slalom thus builds a positional map, accesses the data requested, and places them in a cache. During each subsequent query, Slalom collects statistics regarding the data distribution of the accessed attributes and the average query selectivity to decide whether logical partitioning would benefit performance. If a partition has not reached its *stable* state (i.e., further splitting will not provide benefit), Slalom splits the partition into subsets as described in Section 3.3.1. In state (b), Slalom has already executed some queries and has built a binary cache and a positional map on the accessed attributes. Slalom has decided to logically partition the file into two chunks, of which the first (partition 1) is declared to be in a *stable* state. Slalom checks stable partitions for the existence of indexes; if no index exists, Slalom uses the randomized algorithm described in Section 3.3.2 to decide whether to build one. In state (c), Slalom has executed more queries, and based on the query access pattern it decided index partition 1. In this state, partition 2 of state (b) has been further split into multiple partitions of which partition 2 was declared *stable* and an index was built on it.

3.2.4 Extensibility of Slalom

To address the increasing data format heterogeneity, Slalom queries over a variety of data formats by adding the corresponding parsers and adjusting the online tuner partitioning algorithm. The parser transforms all underlying data to a common representation, which is

then passed to the query engine. This way, Slalom supports multiple data formats by requiring a parser for each input data format (e.g., CSV, JSON, binary). As a common representation, Slalom uses binary tuples stored in fixed length slots. Hence, irrespective to data format, Slalom's binary cache has the same format.

For each new data format, the online tuner applies the same principled techniques of logical horizontal partitioning and indexing, however must be adjusted slightly depending on the format Specifically, when records are stored sequentially (e.g., CSV, binary, XML and JSON), Slalom follows the same approach of partitioning and indexing by creating sequential logical partitions by keeping the first and last byte of each partition within the file. For data formats that store records in a PAX-like format [9] (e.g., parquet, SAM-BAM), the partitioning approach uses as quantum of partitioning the mini-page size rather than a tuple, hence, partitions enclose complete mini-pages. Slalom supports executing queries over CSV, tabular binary and XML files.

3.3 Continuous Partition and Index Tuning

Slalom provides performance enhancements without requiring expensive full data indexing or data file re-organization, all while adapting to workload changes. Slalom uses an online partitioning and indexing tuner to minimize the accessed data by (i) logically partitioning the raw dataset, and (ii) choosing appropriate indexing strategies over each partition. To enable online adaptivity, all decisions that the tuner makes must have minimal computational overhead. The tuner employs a *Partition Manager* which makes all decision considering the partitioning strategy, and an *Index Manager* which makes all decisions considering indexing. This section presents the design of the Partition and Index Managers as well as the mathematical models they are based on.

3.3.1 Raw Data Partitioning

The optimal access path may vary across different parts of a dataset [24]. For example, a filtering predicate may be highly selective in one part of a file, and thus benefit from index-based query evaluation, whereas another file part may be better accessed via a sequential scan. As such, any optimization applied on the entire file may be suboptimal for parts of the file. To this end, the Partition Manager splits the original data into small manageable subsets, having as minimum partition size a physical disk page. The Partition Manager uses horizontal *logical* partitioning as opposed to *physical* partitioning because the latter would require manipulating physical storage – a breaking point for many of the use cases that Slalom targets.

Why Logical Partitions. Slalom uses logical partitioning to virtually break a file into more manageable chunks without physical restructuring. The goal of logical partitioning is twofold: (i) to enable partition filtering, i.e., to group relevant data values together so that they can be skipped for some queries and (ii) to allow for more fine-grained index tuning. The efficiency

of logical partitioning in terms of partition filtering depends mainly on data distribution and performs best with clustered or sorted data. Even in the worst case of uniformly distributed data, although a few partitions will be skipped, the partitioning scheme will facilitate fine-grained indexing. Instead of populating deep B^+ trees that cover the entire dataset, the B^+ trees employed by Slalom are smaller and target only "hot" subsets of the dataset. Thus, Slalom can operate under limited memory budget, has a minimal memory footprint, and provides rapid responses.

The Partition Manager performs partitioning as a by-product of query execution and chooses between two partitioning strategies depending on the cardinality of an attribute. For candidate key attributes, where all tuples have distinct values, the Partition Manager uses *query based partitioning*, whereas for other value distributions, it uses *homogeneous partitioning*. Ideally, the Partition Manager aims to create partitions such that: (i) each partition contains uniformly distributed values, and (ii) partitions are pairwise disjoint (e.g., partition 1 has values 12, 1, 8 and partition 2 has values 19, 13, 30). Uniformly distributed values in a partition enable efficient index access for all values in a partition and creating disjoint partitions improves partition skipping.

Homogenous partitioning

Homogeneous partitioning aims to create partitions with uniformly distributed values and maximize average selectivity within each partition. Increasing query selectivity over partitions implies that for some queries, some of the newly created partitions will contain a high percentage of the final results, whereas other partitions will contain fewer or zero results and will be skippable. Computing the optimal set of contiguous uniformly distributed partitions has exponential complexity, thus is prohibitive for online execution. Instead, to minimize the overhead of partitioning, the Partition Manager iteratively splits a partition into multiple equi-sized partitions. In every iteration, the tuner decides on (i) when to stop splitting and (ii) into how many subsets to split a given partition.

The Partition Manager splits incrementally a partition until it reaches a $stable$ state (i.e., a state where the tuner estimates no more gains can be achieved from further splitting). After each partition split, the tuner relies on two conditions to decide whether a partition has reached a stable state. The tuner considers whether (i) the variance of values in the new partition and the excess kurtosis [97] of the value distribution have become smaller than the variance and kurtosis in the parent partition, and (ii) the number of distinct values has decreased. Specifically, as variance and excess kurtosis decrease, outliers are removed from the partition and the data distribution of the partition in question becomes more uniform. As the number of distinct values per partition iteratively decreases, the probability of partition disjointness increases. If any of these metrics increases or remains stable by partitioning, then the partition is declared stable. We use the combination of variance and excess kurtosis as a metric for uniformity, because their calculation has a constant complexity and can be performed in an incremental fashion during query execution. An alternative would be to use

a histogram or chi square estimators [97], but that would require building a histogram as well as an additional pass over the data.

Making Partitioning Decisions. The number of sub-partitions to which an existing partition is divided, depends on the average selectivity of the past queries accessing the partition and the size of the partition in number of tuples. The goal of the tuner is to maximize selectivity in new partitions. This approach increases the expected number of skipped partitions. We assume that the rows of the partition that have been part of query results within the partition are randomly distributed. We model the partitioning problem as randomly choosing tuples from the partition with the goal to have at least 50% of the new partitions exhibit higher selectivity than the original partition. The intuition is that by decreasing selectivity in a subset of partitions will enhance partition skipping in the rest. The less selective partitions become they contain more result tuples, making the remaining partitions prime candidates for skipping.

We model this problem using the hyper-geometric distribution. Our goal is to choose m partitions by picking randomly n tuples, and we want each partition to contain at least k result tuples. The hyper-geometric distribution is a discrete probability distribution that describes the probability of k random draws in n draws, without replacement. Thus, assuming, that N represents all the tuples in the file, K represents the tuples appearing in the result, and $N-K$ all other tuples. The equation describing the CDF of hypergeometric distribution is the following.

$$P(X \geq k) \approx \sum_{i=k}^{n} \frac{\binom{K}{i}\binom{N-K}{n-i}}{\binom{N}{n}} \tag{3.1}$$

The calculation of the hypergeometric distribution requires the calculation of a factorial and has computational complexity $O(log(log(n \cdot M(n \cdot log n))))$, where $M(n)$ is the complexity of multiplying two n-digit numbers [25]. Such a computational complexity is prohibitively expensive for Slalom as this operation is executed for each query for the majority of partition numerous times and for large partition sizes.

Slalom approximates the hypergeometric distribution using the binomial distribution. Prior work shows that when $p \leq 0.1$ and $N \geq 60$ binomial is a good approximation of hypergeometric [85], and since the sizes of partitions are large in comparison to selectivity Slalom can exploit this observation.

$$P(X \geq k) = \sum_{i=k}^{n} \binom{n}{i} p^i (1-p)^{n-i} \tag{3.2}$$

The binomial distribution requires the calculation of the binomial coefficient $\binom{n}{i}$ which, similarly to the hypergeometric distribution, requires the calculation of factorial. To overcome this problem, we further approximate the binomial coefficient calculation [36]. Specifically we

use the following equation:

$$\binom{n}{k} = \frac{(n/k - 0.5)^k \cdot e^k}{\sqrt{2 \cdot \pi \cdot k}} \tag{3.3}$$

We combine Eq. 3.2 and Eq. 3.3, we use $p = K/N$ and $n = N/m$, and we solve for m to get the the equation that the Partition Manager uses to calculate the number of subpartitions created for every split:

$$m = \frac{N \cdot (sel + \log_b(1 - sel))}{\log_b \frac{\sqrt{2 \cdot \pi \cdot sel \cdot N}}{2}} \qquad \text{where} \qquad b = \frac{e}{sel \cdot (1 - sel)} \tag{3.4}$$

The tuner aims to choose this set of partitions with the minimal computational overhead and number of iterations. The number of distinct values is calculated after each partition is split, whereas the variance and the kurtosis are calculated incrementally, thus the partitioning algorithm has negligible overhead.

Query based partitioning

Query based partitioning targets candidate keys, or attributes that are *implicitly clustered* (e.g., increasing timestamps). For such attributes, homogeneous partitioning will lead to increasingly small partitions as the number of distinct values and variance will be constantly decreasing with smaller partitions. Thus, the tuner decides upon a static number of partitions to split the file. Specifically, the number of partitions is decided based on the selectivity of the first range query using the same mechanism as in homogeneous partitioning. If the partition size is smaller than the physical disk page size, the tuner creates a partition per disk page. By choosing the partitioning approach based on the data distribution, Slalom improves the probability of data skipping and enables fine-grained indexing.

3.3.2 Adaptive Indexing in Slalom

The tuner of Slalom employs the Index Manager to couple logical partitions with appropriate indexes and thus decrease the amount of accessed data. The Index Manager uses *value-existence* and *value-position* indexes; it takes advantage of the capabilities of each category in order to reduce execution overhead and memory footprint. To achieve these goals, the Index Manager enables each partition to have multiple value-existence and value-position indexes.

Value-Existence Indexes. Value-existence indexes are the basis of partition-skipping for Slalom; once a partition has been set as stable, the Index Manager builds a value-existence index over it. Value-existence indexes allow Slalom to avoid accessing some partitions. The Index Manager uses Bloom filters, Bitmaps, and zone maps (min-max values) as value-existence

indexes. Specifically, the Index Manager uses bitmaps only when indexing boolean attributes, because they require a larger memory budget than Bloom Filters for other data types. The Index Manager also uses zone maps on all partitions because they have small memory overhead and provide sufficient information for value-existence on partitions with small value variation. For all other data types, the Index Manager favors Bloom filters because of their high performance and small memory footprint. Specifically, the memory footprint of a Bloom filter has a constant factor, yet it also depends on the number of distinct values it will store and the required false positive probability. To overcome the inherent false positives that characterize Bloom filters, the Index Manager adjusts the Bloom filter's precision by calculating the number of distinct values to be indexed and the optimal number of bytes required to model them [23].

Value-Position Indexes. The Index Manager builds a value-position index (B^+ tree) over a partition to offer fine-grained access to tuples. As value-position indexes are more expensive to construct compared to value-existence indexes, both in terms of memory and time, it is crucial for the index to pay off the building costs in future query performance. The usefulness and performance of an index depend highly on the type and selectivity of queries and the distribution of values in the dataset. Thus, for workloads of shifting locality, the core challenge is deciding *when* to build an index.

When to Build a Value-Position Index. The Index Manager builds a value position index over a partition if it estimates that there will be enough subsequent queries accessing that partition to pay off the investment (in execution time). As the tuner is unaware of the future workload trends, decisions for building indexes are based on the past query access patterns. To make these decisions, the Index Manager uses an online randomized algorithm which considers the cost of indexing the partition ($C_{i_{build}}$), the cost of full partition scan ($C_{i_{fullscan}}$), and the access frequency on the partition (AF_i). These values depend on the data type and the size of the partition, so they are updated accordingly in case of a partition split or an append to the file. The tuner stores the average cost of an access to a file tuple as well as the average cost of an insertion to every index for all data types, and uses these metrics to calculate the cost of accessing and building an index over a partition. In addition, the tuner calculates the cost of an index scan ($C_{i_{indexscan}}$) based on the cost of a full partition scan and the average selectivity. For each future access to the partition, the Index Manager uses these statistics to generate online a probability estimate calculating whether the index will reduce execution time for the rest of the workload. Given this probability, the Index Manager decides whether to build the index.

The Index Manager calculates the index building probability using a randomized algorithm based on the randomized solution of the snoopy caching problem [72]. In the snoopy caching problem, two or more caches share the same memory space which is partitioned into blocks. Each cache writes and reads from the same memory space. When a cache writes to a block, caches that share the block spend 1 bus cycle to get updated. These caches can invalidate the block to avoid the cost of updating. When a cache decides to invalidate a block which

ends up required shortly after, there is a penalty of p cycles. The optimization problem lies in finding when a cache should invalidate and when to update the block. The solution to the index building problem in this work involves a similar decision. The indexing mechanism of the tuner of Slalom decides whether to pay an additional cost per query ("updating a block") or invest in building an index, hoping that the investment will be covered by future requests ("invalidating a block"). Specifically, in cases where the cost of using an index is negligible compared to the cost of full data scan, deciding on index construction can be directly mapped to the snoopy caching problem.

The performance measure of randomized algorithms is the *competitive* ratio (*CR*): the ratio between the expected cost incurred when the online algorithm is used and that of an optimal offline algorithm that we assume has full knowledge of the future. When index access cost is negligible, the randomized algorithm of the tuner guarantees optimal CR ($\frac{e}{e-1}$). The tuner uses a randomized algorithm in order to avoid the high complexity of what-if analysis [104] and to improve the competitive ratio offered by the deterministic solutions [26].

Cost Model. Assume query workload W. At a given query q of the workload, a partition is in one of two states: it either has an index or it does not. A state is characterized by the pair (C_{build}, C_{use}) where C_{build} is the cost to enter the state (e.g., build the index) and C_{use} the cost to use the state (e.g., use the index). The initial state is the state with no index (i.e., full scan) $(C_{build,fs}, C_{use,fs})$ where $C_{build,fs} = 0$. In the second state $(C_{build,idx}, C_{use,idx})$, the system has an index . We assume that the relation between the costs for the two states is $C_{build,idx} > C_{build,fs}$ and $C_{use,idx} < C_{use,fs}$ and $C_{build,idx} > C_{use,fs}$.

Given a partition i, the index building cost over that partition ($C_{i_{build}}$), the full partition scan cost ($C_{i_{fullscan}}$), the index partition scan cost ($C_{i_{indexscan}}$) and a sequence of queries Q : $[q_1, \ldots, q_T]$ accessing the partition. Assume that q_T is the last query that accesses the partition (*and is not known*). At the arrival time of $q_k, k < T$, we want to decide whether the Index Manager should build the index or perform full scan over the partition to answer the query.

To make the decision we need a probability estimate p_i for building the index at moment i based on the costs of building the index or not. In order to calculate p_i we initially define the overall expected execution cost of the randomized algorithm that depends on the probability p_i. The expected cost E comprises three parts:

i. the cost of using the index, which corresponds to the case where the index has already been built.

ii. the cost of queries doing full partition scan, which corresponds to the case for which the index has not be built.

iii. the cost of building the index, which corresponds to the case where the building of the index will take place at time i. Index construction takes place as a by-product of query execution and includes the cost of the current query.

$$E = \sum_{i=1}^{T} \Big(\sum_{j=1}^{i-1} p_j \cdot C_{use,idx} + \Big(1 - \sum_{j=1}^{i-1} p_j\Big) \cdot \Big(p_i \cdot C_{build,idx} + (1-p_i) \cdot C_{use,fs}\Big)\Big)$$

Knowing the expected cost we minimize and we solve for p_i. We exchange $C_{build,idx}$ with $C_{use,fs} + \delta$ as building the index will cost at least as much as a full scan.

$$E = T \cdot C_{use,fs} - \Big(C_{use,fs} - C_{use,idx}\Big) \cdot \Big(\sum_{i=1}^{T} \sum_{j=1}^{i-1} p_j\Big) + \delta \cdot \Big(\sum_{i=1}^{T} p_i - \sum_{i=1}^{T} p_i \cdot \sum_{j=1}^{i-1} p_j\Big) \tag{3.5}$$

We take the first partial derivative of this formula for p_i.

$$\frac{\partial E}{\partial p_i} = -\Big(C_{use,fs} - C_{use,idx}\Big) \cdot \frac{\partial\Big(\sum_{i=1}^{T}\sum_{j=1}^{i-1} p_j\Big)}{\partial p_i} + \delta \cdot \Big(\frac{\partial \sum_{i=1}^{T} p_i}{\partial p_i} - \frac{\partial\Big(\sum_{i=1}^{T} p_i \cdot \sum_{j=1}^{i-1} p_j\Big)}{\partial p_i}\Big) \tag{3.6}$$

We calculate that:

$$\frac{\partial\Big(\sum_{i=1}^{T}\sum_{j=1}^{i-1} p_j\Big)}{\partial p_i} = (T-i) \tag{3.7}$$

and

$$\frac{\partial\Big(\sum_{i=1}^{T} p_i \cdot \sum_{j=1}^{i-1} p_j\Big)}{\partial p_i} = \sum_{j=1}^{T-1} p_j - p_i \tag{3.8}$$

Thus, the final derivative becomes:

$$\frac{\partial E}{\partial p_i} = -\Big(C_{use,fs} - C_{use,idx}\Big) \cdot \Big(T-i\Big) + \delta \cdot \Big(1 - \sum_{j=1}^{T-1} p_j - p_i\Big) \tag{3.9}$$

To minimize the Expected cost we solve the equation and we solve for p_i.

$$\frac{\partial E}{\partial p_i} = 0 => p_i = \frac{C_{use,fs} - C_{use,idx}}{\delta} \cdot (T-i) - \Big(1 - \sum_{j=1}^{T-1} p_j\Big) \tag{3.10}$$

The final probability equation is:

$$p_i = \frac{C_{use,fs} - C_{use,idx}}{C_{build,idx} - C_{use,fs}} \cdot (T-i) - \Big(1 - \sum_{j=1}^{i-1} p_j\Big) \tag{3.11}$$

Based on our model, performing a full scan over the complete data file should be always cheaper than an index access and the amortized extra cost of building the index (over T queries).

Eviction Policy. The tuner works within a predefined memory budget to minimize memory overhead. If the memory budget is fully consumed and the Index Manager attempts to build a new index, then it defers index construction for the next query and searches indexes to drop to make the necessary space available. The Index Manager keeps all value-existence indexes once built, because their size is minimal and they are the basis of partition skipping. Furthermore, the Index Manager prioritizes binary caches over indexes, because (i) using a cache improves the performance of all queries accessing a partition, and (ii) accessing the raw data file is typically more expensive than rebuilding an index for large partitions. Deciding which indexes from which partitions to drop is based on index size ($Size_{index_i}$), number of queries since last access (LA_i), and average selectivity (sel_i) in a partition. To compute the set of indexes to drop, the Index Manager uses a greedy algorithm which gathers the least accessed indexes with cumulative size ($\sum_i Size_{index_i}$) equal to the size of the new index. Specifically, to discover the least accessed indexes, the Index Manager keeps a bitmap of accesses for each partition. During a query predicate evaluation on a partition and depending on whether the current query touches the partition, the Index Manager shifts the partition's bitmap to the left and appends a bit to it: 1 (yes) or 0 (no). When calculating the candidate indexes to drop, the Index Manager uses SIMD instructions to evaluate the set of least accessed partitions. Specifically, each bitmap is an 8-byte unsigned integer which stores the past 64 queries. In a 256-byte wide CPU register, the Index Manager uses a bitmask operation to check the occupancy of 32 partitions simultaneously. When all indexes are used with the same frequency, the tuner uses the average selectivity of queries on each partition as a tie-breaker condition. The less selective queries are, the smaller the gap between index and full scan performance, therefore the Index Manager victimizes partitions touched by non-selective queries.

3.3.3 Handling File Updates

Slalom supports both append-like and in-place updates directly over the raw data file and ensures consistent results. In order to achieve efficient data access and correct results despite updates, Slalom continuously monitors the queried files for any write operation and stores summaries of the queried files representing their current state. If a file is updated, Slalom compares its existing summary with the stored state identifies the changes, and updates any dependent data structures.

In this section, we describe in detail how Slalom: (i) monitors its input files for updates at real-time, (ii) calculates and stores a summary of the most recent consistent state for reference, (iii) identifies the updated file subsets, and (iv) updates its internal data structures.

Monitoring Files

In order to recognize whether an input file has been updated by another application (e.g., vim), Slalom uses OS support (i.e., inotify [76]). Specifically, Slalom initializes a watchdog, over the queried file, which is triggered when the file is written upon and adds a log entry into a queue. This queue contains all updates that have not been addressed by Slalom yet. Slalom checks the queue for new updates both at the beginning of every query as well as during execution. During query execution, Slalom checks whether already scanned data received any updates. If this is the case, the query is re-executed to ensure consistency and correctness of the final query result, that is to ensure that the query is answered by a single file version.

Calculating and Storing State

In order to be able to discover the updated rows in the file and the type of update (append or in-place), Slalom exploits its logical partitioning scheme. For each partition, Slalom stores a checksum encoding the contents within that partition and the starting and ending positions of the partition in the file. This information is generated during the first pass after the partition creation. It is sufficient to identify the existence of an update within a partition as it summarize the size as well as the content of each partition. As the checksum calculation is part of the critical path of query execution it increases the query runtime. To alleviate this cost, Slalom exploits specialized hardware that offers high throughput in checksum calculation. Thus, depending on the available hardware, Slalom uses different checksums.

By default, Slalom uses sequential 64-byte MD5 checksum. Checksum calculations are compute-heavy, hence, Slalom migrates when possible such calculations to a General Purpose Graphics Processing Unit (GPGPU) using an MD5 CUDA implementation. Finally, given smaller logical partitions, Slalom takes advantage of the 32-bit Cyclic Redundancy Checking (CRC) on-board chip to calculate a checksum.

MD5 Algorithm. MD5 [103] is a cryptographic hash function, that is widely used data integrity verification checksum [112]. Given input of arbitrary size, MD5 algorithm produces a 128-bit output, which is usually represented in 32 hexadecimal digits. MD5 uses four non-linear functions and it can deal with data of arbitrary length. MD5 serves as a good candidate for detecting file updates, however, calculating it on a single CPU can be prohibitively expensive. Thus, we design a parallelization scheme for MD5.

MD5 is an irreversible transformation of a set of data of any length into a hash value of 128-bit length. MD5 is a consecutive processing method as the original algorithm processes the input data incrementally in 512-bit groups and combines them with the result coming from the processing of prior groups. To parallelize the computation of MD5, we need to enable the parallel computation of different portions of the checksum. Initially, we divide the input data into small data blocks with the same size. Subsequently, we perform the standard MD5 algorithm on each data block, in parallel, and we store the calculated checksums

in sequence. The resulting checksums are combined until the result is 128-bit long. The checksum computed by this approach is not identical to the standard MD5 checksum, however, has equal encryption strength [59].

We implement the parallel MD5 with NVIDIA CUDA on GPU,which is inherently suitable for multi-threading. CUDA provides a convenient programming interface that extends the C language and allows programmers to write C functions as GPU kernels, that will be executed by multiple CUDA threads.

CRC. Cyclic Redundancy Codes are used to mostly detect errors in network packets [99]. As this operation is latency-sensitive, modern processors have added the CPU instructions `_mm_crc32_u64` for calculating 32-bit CRC codes to its SSE4.2 instruction set. Originally, a CRC code is calculated as follows: to obtain m-bit CRC code, the n-bit input data is first appended with m zeros. Then it is `XOR`ed with a polynomial divisor of the size of $(n+1)$ bit from left to right. The last m bits are the final resulting code.

Typically a n-bit CRC applied to a data block of arbitrary length will detect a single error burst that is not longer than n bits and will detect a fraction $\frac{1}{(1-2^{-n})}$ of all longer error bursts. As partitions used by Slalom can be of arbitrary size, Slalom calculates the 32-bit CRC value for each 1024-byte block in the partition and then adds up all computed values to give the final verification code. This code has the same detection ability, namely detecting changes no longer than 4 bytes.

Recognizing Update Type and Updating Data Structures

To provide efficient data access, Slalom builds a set of data structures which are based on the structure of the queried file. Updates may change that structure and make the data structures obsolete. Specifically, indexes and positional maps are sensitive to the specific location of attributes and number of tuples within the file. Similarly, caches and Bloom filters become obsolete with any change in a partition. To overcome this issue, Slalom updates its data structures depending on the update type. To identify the type of update, Slalom compares the current state of each partition with the stored one. Thus, Slalom checks whether starting and ending characters of the partition have changed or if the checksum has changed. If the state of each partition matches with the existing one, then the update type is append. Otherwise, it is an in-place update.

Append-like Updates. Slalom supports updates in an append-like scenario without disturbing query execution by dynamically extending auxiliary data structures. In append-like scenarios, Slalom creates a new partition at the end of the file to accommodate the new data. Depending on the partitioning approach, Slalom either accumulates updates to create partitions of equal size (i.e., query-based partitioning) or dynamically repartitions the fresh data. Once Slalom has organized the new data in partitions, it treats them similarly to a first time input. Thus,

during the first query after an update, Slalom builds binary caches and positional maps over the new data. When the new partitions are declared $stable$, Slalom builds indexes on top of them.

In-place Updates. In-place updates correspond to random changes in the file made by another application, such as updating values of specific fields or adding additional rows in the middle of the file. In-place updates are more challenging, especially when considering the case of the positional map and indexes. A change in a position of an attribute in the data file might require significant reorganization in all generated data structures.

Updating Positional Maps: To update the positional map for a modified partition, Slalom scans each field character by character each field to narrow down the updated parts. Once the updated section is identified, Slalom stores the difference in byte offsets between the old and new fields into a *delta list*. All new changes are appended to the list and any possible changes in previous offset differences are being integrated as well. The delta list adds computational overhead when using the positional map because, for every access Slalom must check the delta list whether the position has been altered by an update. As the delta list grows, the complexity of position computation is growing as well. Thus, to reduce the query cost, the delta list is incorporated into the original positional map every 10 updates. Specifically, to incorporate the delta list into the positional map, Slalom scans over the delta list and adds the offsets to the existing indexes in the positional map. This way, it does not have to completely reconstruct the positional map while reducing the delta list.

Updating caches and indexes: In order to maintain a minimal memory footprint, Slalom does not store a replica of the original file to be able retrieve old values for each updated field. Hence, Slalom is unable to update indexes and caches. Rather, it invalidates and re-builds them.

3.4 Experimental Evaluation

In this section, we present an analysis of Slalom. We analyze its partitioning and indexing algorithm, and compare it against state-of-the-art systems over both synthetic and real life workloads.

Methodology. We compare Slalom against DBMS-X, a commercial state-of-the-art in-memory DBMS that stores records in a row oriented manner and the open-source DBMS PostgreSQL (version 9.3). We use DBMS-X and PostgreSQL with two different configurations: (i) Fully-loaded tables and (ii) Fully-loaded, indexed tables. We also compare Slalom with the in-situ DBMS PostgresRaw [10]. PostgresRaw is an implementation of NoDB [10] over PostgreSQL; PostgresRaw avoids data loading and executes queries by performing full scans over CSV files. In addition, PostgresRaw builds positional maps on-the-fly to reduce parsing and tokenization costs. Besides positional maps, PostgresRaw uses caching structures to hold previously ac-

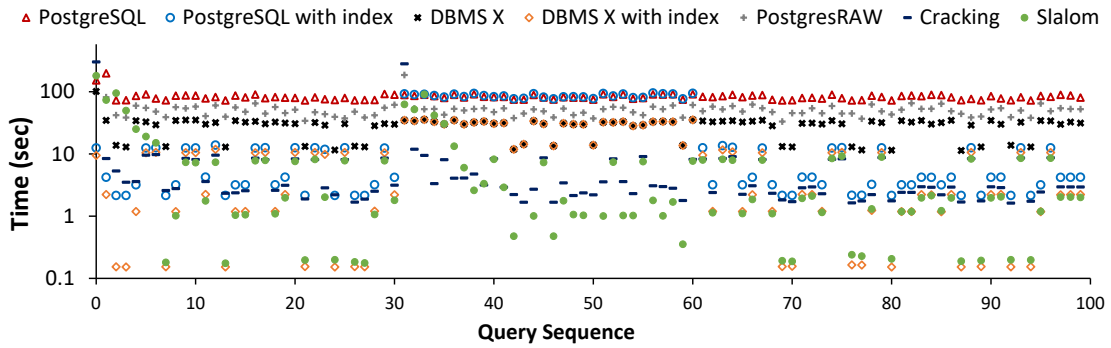

Figure 3.4 – Sequence of 100 queries. Slalom dynamically refines its indexes to reach the performance of an index over loaded data.

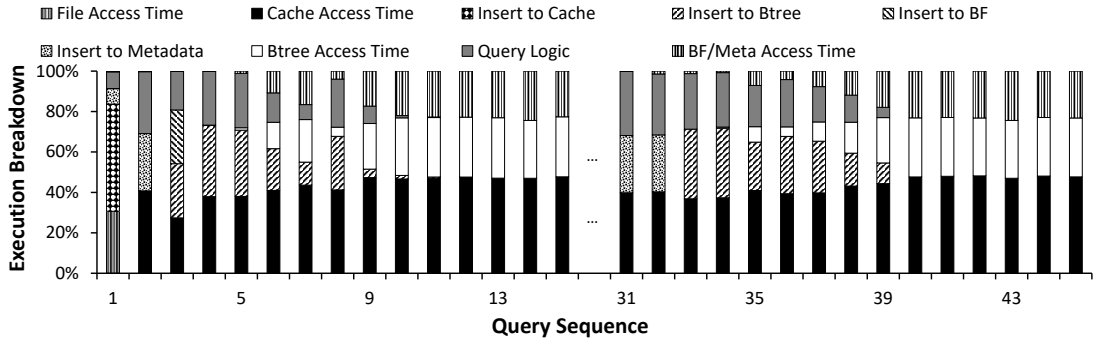

Figure 3.5 – A breakdown of the operations taking place for Slalom during the execution of a subset of the 100 point query sequence.

cessed data in a binary format. Furthermore, to compare Slalom with other adaptive indexing techniques we integrate into Slalom two variations of database cracking: (i) standard cracking [61] and (ii) the MDD1R variant of stochastic cracking [55]. We chose MDD1R as it showed the best overall performance in [105]. We integrated the cracking techniques by disabling the Slalom tuner and setting cracking as the sole access path. Thus, Slalom and cracking use the same execution engine and have the same data access overheads.

Slalom's query executor pushes predicate evaluation down to the access path operators for early tuple filtering and results are pipelined to the other operators of a query (e.g., joins). Thus, in our analysis, we focus on scan intensive queries. We use select-project-aggregate queries to minimize the number of tuples returned and avoid any overhead from the result tuple output that might affect the measured times. Unless otherwise stated, the queries are of the following template ($OP : \{<, >, =\}$):

```
SELECT agg(A), agg(B), ..., agg(N) FROM R
WHERE A OP X (AND A OP Y)
```

Experimental Setup. The experiments are conducted in a Sandy Bridge server with a dual socket Intel(R) Xeon(R) CPU E5-2660 (8 cores per socket @ 2.20 Ghz), equipped with 64KB L1 cache and 256KB L2 cache per core, 20MB L3 cache shared, and 128GB RAM running Red Hat Enterprise Linux 6.5 (Santiago - 64 bit) with kernel version 2.6.32. The server is equipped with a RAID-0 of 7 250GB 7500 RPM SATA disks.

3.4.1 Adapting to Workload Shifts

Slalom adapts efficiently to workload shifts despite changes in (i) data distribution, (ii) query selectivity, and (iii) query locality - both vertical (i.e., different attributes) and horizontal (i.e., different records). We demonstrate the adaptivity experimentally by executing a dynamic workload with varying selectivity and access patterns over a synthetic dataset.

Methodology. To emulate the worst possible scenario for Slalom, we use a relation of 640 million tuples (59GB), where each tuple comprises of 25 unsigned integer attributes with uniformly distributed values ranging from 0 to 1000. Slalom is unable to find a value clustering in the file because all values are uniformly distributed, thus Slalom applies homogeneous partitioning. Slalom, cracking, and PostgresRaw operate over the CSV data representation, whereas PostgreSQL and DBMS-X load the raw data prior to querying. In this experiment, we limit the index memory budget for Slalom to 5GB and the cache budget to 10GB. All other systems are free to use all available memory. Specifically, for this experiment, DBMS-X required 98GB of RAM to load and fully build the index.

We execute a sequence of 1000 point and range select-project-aggregation queries following the template from Section 3.4. The predicate value is randomly selected from the domain of the attribute. Point query selectivity is 0.1% and range query selectivity varies from 0.5% to 5%. To emulate workload shifts and examine system adaptivity, in every 100 queries, queries 1-30 and 61-100 use a predicate on the first attribute of the relation and queries 31-60 use a predicate on the second attribute.

The indexed variations of PostgreSQL and DBMS-X build a clustered index only on the first attribute. It is possible to build indexes on more columns for PostgreSQL and DBMS-X, however, it requires additional resources and increases data-to-query time. In addition, choosing which attributes to index, requires a priori knowledge of the query workload, which is unavailable in the dynamic scenarios that Slalom considers. Indicatively, building an secondary index on a column for PostgreSQL for our experiment takes ~25 minutes. Thus, by the time PostgreSQL finishes indexing, Slalom will have finished executing the workload (Figure 3.6).

Slalom Convergence. Figure 3.4 shows the response time of each query of the workload for the different system configurations. For clarity, we present the results for the first 100 queries. To emulate the state of DBMS systems immediately after loading, all systems run from a hot

state where data is resting in the OS caches. Figure 3.4 plots only query execution time and does not show data loading or index building for PostgreSQL and DBMS-X.

The runtime for the first query of Slalom is 20× slower than its average query time, because during that query it builds a positional map and a binary cache. In subsequent queries (queries 2-7), Slalom iteratively partitions the dataset and builds B^+ trees. After the initial set of queries (queries 1-6), Slalom has comparable performance to that of PostgreSQL over fully indexed data. During the third query, multiple partitions stabilize simultaneously, thus Slalom builds many B^+ tree and Bloom filter indexes, adding considerable overhead. When Slalom converges to its final state, its performance is comparable to that of the indexed DBMS-X. When the queried attribute changes (query 31), Slalom starts partitioning and building indexes on the new attribute. After query 60, when the workload filters data based on the first attribute again, where the partitioning is already stable, Slalom re-uses the pre-existing indexes.

PostgreSQL with no indexes demonstrates a stable execution time as it has to scan all data pages of the loaded database regardless of the result size. Due to the queries being very selective, when an index is available for PostgreSQL, the response times are ~9× lower when queries touch the indexed attribute. DBMS-X keeps all data in memory and uses memory-friendly data structures, so it performs on average 3× better than PostgreSQL. The difference in performance varies with query selectivity. In highly selective queries, DBMS-X is more efficient in data access whereas for less selective queries the performance gap is smaller. Furthermore, for very selective queries, indexed DBMS-X is more efficient than Slalom as its single B^+ tree traverses very few results nodes.

During query 1, PostgresRaw builds auxiliary structures (cache, positional map) and takes 3× more time (180 sec) than its average query run time. PostgresRaw becomes faster than the unindexed PostgreSQL variation as its scan operators use vector-based (SIMD) instructions and exploit compact caching structures.

Similarly, during query 1, cracking builds a binary cache and populates the cracker column it uses for incremental indexing. The runtime of its first query is 4× slower than the average query time for PostgreSQL without indexes. When it touches a different attribute (query 31) it also populates a cracker column for the second attribute. Despite the high initialization cost, cracking converges efficiently and reaches its final response time after the fourth query. The randomness in the workload benefits cracking as it splits the domain into increasingly smaller pieces. After converging, cracking performance is comparable to the PostgreSQL with index. Slalom requires more queries to converge than cracking. However, after it converges, Slalom is ~2× faster than cracking. This difference stems from cracking execution overheads. cracking sorts the resulting tuples based on their memory location and enforces sequential memory access. This sorting operation adds an overhead, especially for less selective queries.

Execution Breakdown. Slalom aims to build efficient access paths with minimal overhead. Figure 3.5 presents the breakdown of query execution for the same experiment as before. For

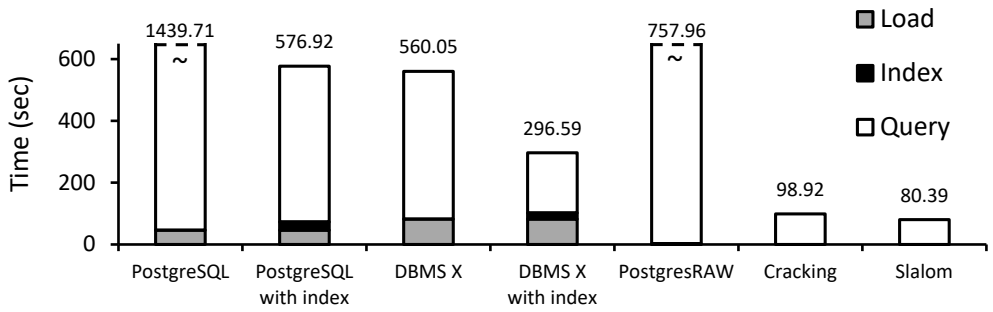

Figure 3.6 – Sequence of 1000 queries. Slalom does not incur loading cost and dynamically builds indexes.

clarity, we present only queries Q1-15 and Q31-45 as Q16-30 show the same pattern as Q11-15. Queries Q1-15 have a predicate on the first attribute and queries Q31-45 have a predicate on the second attribute.

During the first query, Slalom scans through the original file and creates the cache. During Q2 and Q3 Slalom is actively partitioning the file and collects data statistics (i.e., distinct value counts) per partition; Slalom bases the further partitioning and indexing decisions on these statistics. Statistics gathering cost is represented in Figure 3.5 as "Insert to Metadata". During queries Q2 and Q3, as the partitioning scheme stabilizes, Slalom builds Bloom filters and B^+ trees. Q3 is the last query executed using a full partition scan, and since it also incurs the cost of index construction there is a local peak in execution time. During Q4 through Q8, Slalom increasingly improves performance by building new indexes. After Q31, the queries use the second attribute of the relation in the predicate, thus Slalom repeats the process of partitioning and index construction. In total, even after workload shifts, Slalom converges into using index-based access paths over converted binary data.

Full Workload: From Raw Data to Results. Figure 3.6 presents the full workload of 1000 queries, this time starting with cold OS caches and no loaded data to include the cost of the first access to raw data files for all systems. We plot the aggregate execution time for all approaches described earlier, including the loading and indexing costs for PostgreSQL and DBMS-X.

PostgresRaw, Slalom, and cracking incur no loading and indexing cost, and start answering queries before the other DBMS load data and before the indexed approaches finish index building. Unindexed PostgreSQL incurs data loading cost as well as a total query aggregate greater than PostgresRaw. Indexed PostgreSQL incurs both indexing and data loading cost, and due to some queries touching a non-indexed attribute, its aggregate query time is greater than the one of Slalom. Unindexed DBMS-X incurs loading cost; however, thanks to its main memory-friendly data structures and execution engine, it is faster than the disk-based engine of PostgreSQL.

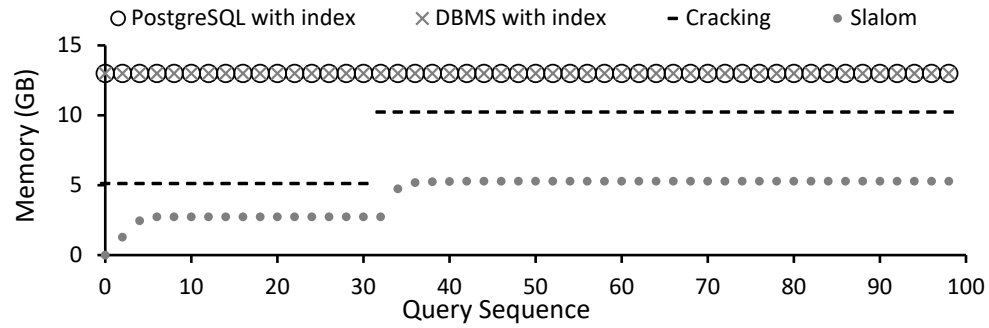

Figure 3.7 – Memory consumption of Slalom vs. a single fully-built B$^+$ tree for PostgreSQL and DBMS-X. Slalom uses less memory because its indexes only target specific areas of a raw file.

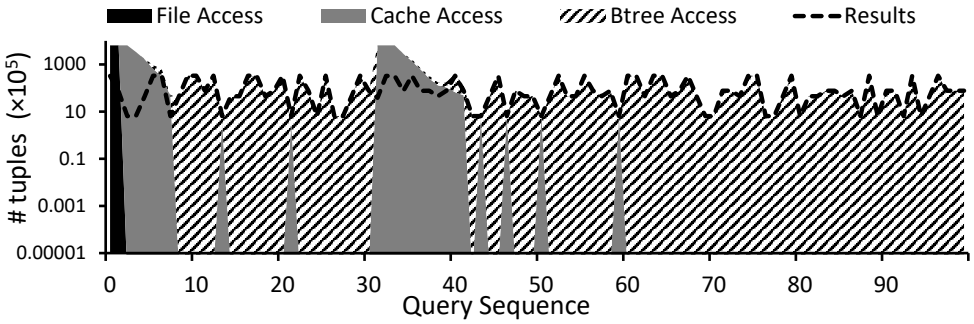

Figure 3.8 – Number of accessed tuples using file, cache or B$^+$ tree corresponding to the 100 queries of synthetic workload.

After adaptively building the necessary indexes, Sla- lom has comparable performance with a conventional DBMS which uses indexes. cracking converges quickly and adapts to the workload efficiently. However, creating the cracker columns incurs a significant cost. Overall, cracking and Slalom offer comparable raw-data-to-results response time for this workload while, Slalom requires 0.5× memory. We compare in detail cracking and Slalom in Section 3.4.3.

Memory Consumption. Figure 3.7 plots the memory consumption of (i) the fully built indexes used for DBMS-X and PostgreSQL, (ii) the cracker columns for cracking, and (iii) the indexes of Slalom. Figure 3.7 excludes the size of the caches used by Slalom and cracking or the space required by DBMS-X after loading. The traditional DBMS require significantly more space for their indexes. Orthogonally to the index memory budget, DBMS-X required 98GB of memory in total, whereas the cache of Slalom required 9.7GB. cracking builds its cracker columns immediately when accessing a new attribute. The cracker column requires storing the original column values as well as pointers to the data, thus it has a large memory footprint even for low value cardinality. Regarding the indexes of Slalom, when the focus shifts to another filtering attribute (Q31), Slalom increases its memory consumption, as during Q31-34 it creates logical

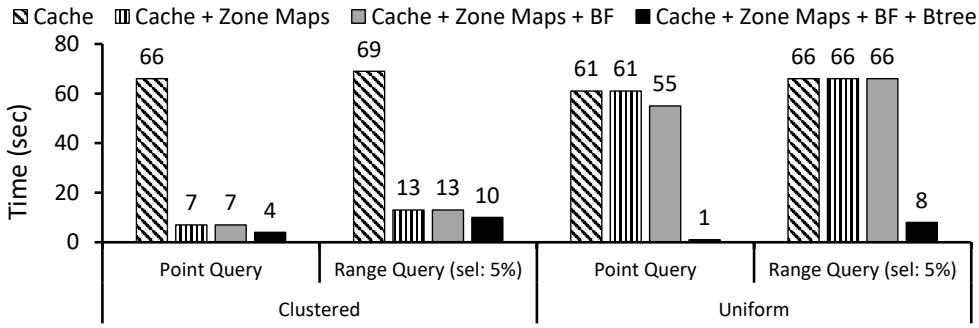

Figure 3.9 – The effect of different indexes on point and range queries over uniform and clustered datasets.

partitions and builds Bloom filters and B$^+$ tree indexes on the newly accessed attribute. By building and keeping only the necessary indexes for a query sequence, Slalom strikes a balance between query performance and memory utilization.

Minimizing Data Access. The performance gains of Slalom are a combination of data skipping based on partitioning, value-existence indexes, and value-position indexes, all of which minimize the number of tuples Slalom has to access. Figure 3.8 presents the number of tuples that Slalom accesses for each query in this experiment. We observe that as the partitioning and indexing schemes of Slalom converge, the number of excess tuples accessed is reduced. Since the attribute participating in the filtering predicate of queries Q31-60 has been cached, Slalom accesses the raw data file only during the first query. Slalom serves the rest of the queries utilizing only the binary cache and indexes. For the majority of queries, Slalom responds using an index scan. However there are queries where it responds using a combination of partition scan and index scan.

Figure 3.9 presents how the minimized data access translates to reduced response time and the efficiency of data skipping and indexing for different data distribution and different query types. Specifically, it presents the effect of zone maps, Bloom filters and B$^+$ trees on query performance for point queries and range queries with 5% selectivity over uniform and clustered datasets. The clustered dataset contains mutually disjointed partitions (i.e., subsets of the file contain values which do not appear in the rest of the file). The workload used is the same used for Figure 3.4. Zone maps are used for both range and point queries and are most effective when used over clustered data. Specifically, they offer a ~9× better performance than full cache scan. Bloom filters are useful only for point queries. As the datasets have values in the domain [1,1000], point queries have low selectivity making Bloom filters ineffective. Finally, B$^+$ trees improve performance for both range and point queries. The effect of B$^+$ tree is seen mostly for uniform data where partition skipping is less effective. Slalom stores all indexes in-memory, thus by skipping a partition, Slalom avoids full access of the partition and reduces memory access or disk I/O if the partition is cached or not respectively.

Figure 3.10 – Slalom performance using different memory budgets.

Summary. We compare Slalom against (i) a state-of-the-art in-situ querying approach, (ii) a state-of-the-art adaptive indexing technique, (iii) a traditional DBMS, and (iv) a state-of-the-art in-memory DBMS. Slalom gracefully adapts to workload shifts using an adaptive algorithm with negligible execution overhead. Slalom offers performance comparable with a DBMS which uses indexes, while also being more conservative in memory space utilization.

3.4.2 Working Under Memory Constraints

As described in Section **??**, Slalom efficiently uses the available memory budget to keep the most beneficial auxiliary structures. We show this experimentally by executing the same workload under various memory utilization constraints. We run the 20 first queries – a mix of point and range queries. We consider three memory budget configurations with 10GB, 12GB and 14GB of available memory, respectively. The budget includes both indexes and caches.

Figure 3.10 presents the query execution times for the workload given the three different memory budgets. The three memory configurations build a binary cache and create the same logical partitioning. Slalom requires 13.5GB in total for this experiment; given an 14GB memory budget, it can build all necessary indexes, leading to the best performance for the workload. For the 10GB and 12GB memory budgets, there is insufficient space to build all necessary indexes, thus these configurations experience a performance drop. We observe that configurations with 10GB and 12GB memory budgets outperform the configuration with 14GB of memory budget for individual queries (i.e., Q3 and Q5). The reason is that the memory-limited configurations build fewer B^+ trees during these queries than the configuration with 14GB of available memory. However, future queries benefit from additional B^+ trees, amortizing the extra overhead over a sequence of queries.

Figure 3.11 presents the breakdown of memory allocation for the same query sequence when Slalom is given a 12GB memory budget. We consider the space required for storing caches, B^+ trees and Bloom filters. The footprint of the statistics and metadata Slalom collects for the cost model and zone maps is negligible, thus we exclude them from the breakdown. Slalom initially builds the binary cache, and logically partitions the data until some partitions become

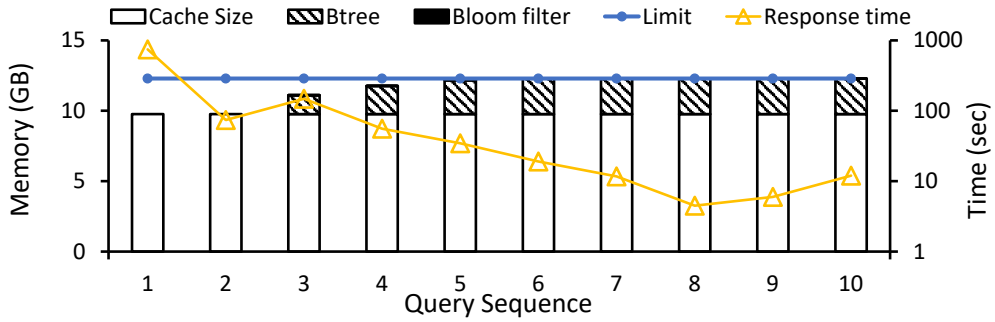

Figure 3.11 – Slalom memory allocation (12GB memory budget).

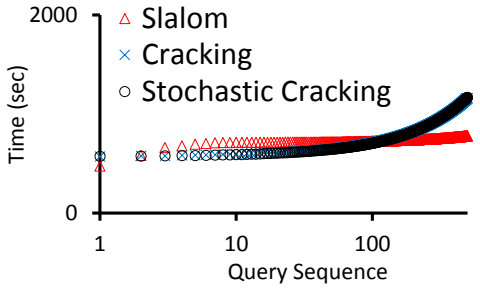

Figure 3.12 – Random/Uniform data

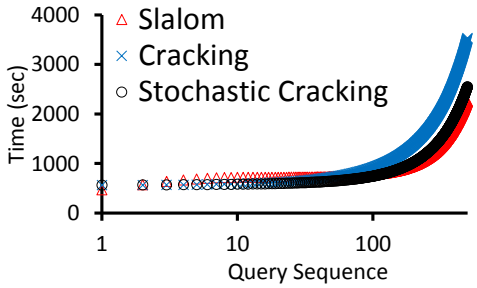

Figure 3.13 – Zoom In Alt./Uniform data

stable (Q1, Q2). At queries Q3, Q4, and Q5 Slalom starts building B^+ trees, and it converges to a stable state at query Q7 where all required indexes are built. Thus, from Q7-Q10 Slalom stabilizes performance. Overall, this experiment shows that Slalom can operate under limited memory budget gracefully managing the available resources to improve query execution performance.

3.4.3 Adaptivity Efficiency

Slalom adapts to query workloads as efficiently as state-of-the-art adaptive indexing techniques while working with less memory. Furthermore, it exploits any potential data clustering to further improve its performance. We demonstrate this by executing a variety of workloads. We use datasets of 480M tuples (55GB on disk); each tuple comprises 25 unsigned integer attributes whose values belong to the domain $[1, 10000]$. Queries in all workloads have equal selectivity to alleviate the noise from data access; all queries have 0.1% selectivity, i.e., select 10 consecutive values.

```
SELECT agg(A), ..., agg(E) FROM R
WHERE R.A >= low AND low+10 <= R.A
```

Methodology. Motivated by related work [105], we compare Slalom against cracking and

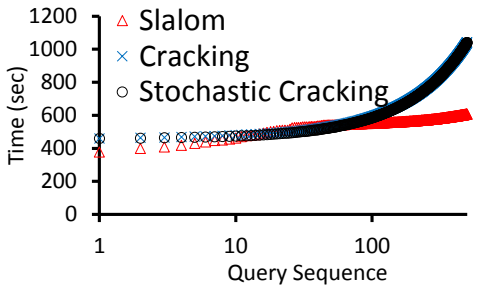

Figure 3.14 – Random/Clustered data

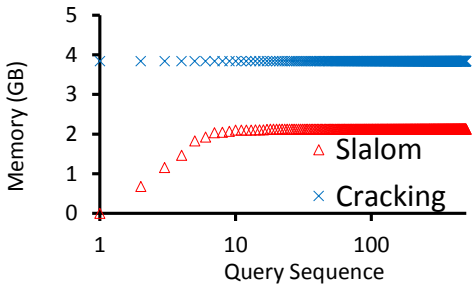

Figure 3.15 – Memory on Random/Uniform data

stochastic cracking in three cases.

Random workload over Uniform dataset. We execute a sequence of range queries which access random ranges throughout the domain to emulate the best case scenario for cracking. As subsequent queries filter on random values and the data is uniformly distributed in the file, cracking converges and minimizes data access.

"Zoom In Alternate" over Uniform dataset. To emulate the effect of patterned accesses, we execute a sequence of queries that access either part of the domain in alternate, i.e., first query: [1,10], second query: [9991,10000], third query: [11,20], etc. This access pattern is one of the scenarios where the original cracking algorithm underperforms [55]. Splits are only query-driven, and every query splits data into a small piece and the rest of the file. Thus, the improvements in performance with subsequent queries are minimal. Stochastic cracking alleviates the effect of patterned accesses by splitting in more pieces apart from the ones based on queries.

Random workload over Clustered dataset. This setup examines how adaptive indexing techniques perform on datasets where certain data values are clustered together, for example, data clustered on timestamp or sorted data. The clustered dataset we use in the experiment contains mutually disjoint partitions, i.e., subsets of the file contain specific values which appear solely in those locations and do not appear in the rest of the file.

Figure 3.12 demonstrates the cumulative execution time for cracking, stochastic cracking and Slalom for the random workload over uniform data. All approaches start from a cold state, thus during the first query they parse the raw data file and build a binary cache. Stochastic cracking and cracking incur an additional cost of cracker column initialization during the first query, but reduce execution time with every subsequent query. During the first three queries, Slalom creates its partitions; during the following 6 queries, Slalom builds the required indexes, and finally converges to a stable state at query 10. Due to its fine-grained indexing and local memory accesses, Slalom provides ~8× lower response time than cracking and their cumulative execution time is equalized during query 113. Furthermore, Figure 3.15 demonstrates the memory consumption of the cracking approaches and Slalom for the same

Table 3.2 – Cost of each phase of a smart-meter workload.

System	Loading	Index Build	Queries	Total
Slalom	0 sec	0 sec	4301 sec	4301 sec
Cracking	0 sec	0 sec	6370 sec	6370 sec
PostgresRaw	0 sec	0 sec	10077 sec	10077 sec
PostgresSQL (with index)	2559 sec	1449 sec	9058 sec	13066 sec
PostgreSQL (no index)	2559 sec	0 sec	15379 sec	17938 sec
DBMS-X (with index)	6540 sec	1207 sec	3881 sec	11628 sec
DBMS-X (no index)	6540 sec	0 sec	5243 sec	11783 sec

experiment. The cracking approaches have the same memory footprint; they both duplicate the full indexed column along with pointers to the original data. On the other hand, the cache-conscious B^+ trees of Slalom stores only the distinct values along with the positions of each value, thus reducing the memory footprint. In addition, Slalom allocates space for its indexes gradually, offering efficient query execution even with limited resources.

Figure 3.13 shows the cumulative execution time for cracking, stochastic cracking, and Slalom for the "Zoom In Alternate" workload over uniform data. cracking needs more queries to converge to its final state as it is cracking only based on query-driven values. Stochastic cracking converges faster because it cracks based on more values except the ones found in queries. Slalom uses a combination of data and query driven optimizations. Slalom requires an increased investment during the initial queries to create its partitioning scheme and index the partitions, but ends up providing 7× lower response time, and equalizes cumulative execution time with cracking at query 53 and stochastic cracking at query 128.

Figure 3.14 presents the cumulative execution time of cracking, stochastic cracking and Slalom for the random workload over implicitly clustered data. In this situation, Slalom exploits the clustering of the underlying data early on (from the second query) and skips the majority of data. For the accessed partitions, Slalom builds indexes to further reduce access time. Similarly to Figure 3.12, the cracking approaches crack only based on the queries and are agnostic to the physical organization of the dataset.

Summary. We compare Slalom with cracking and stochastic cracking. Slalom converges comparably to the best cracking variation when querying uniform data over both random and "Zoom In Alternate" workloads. Furthermore, when Slalom operates over clustered data, it exploits the physical data organization and provides minimal data-to-query time. Finally, as Slalom builds indexes gradually and judiciously, it requires less memory than the cracking approaches, and it can operate under a strict memory budget.

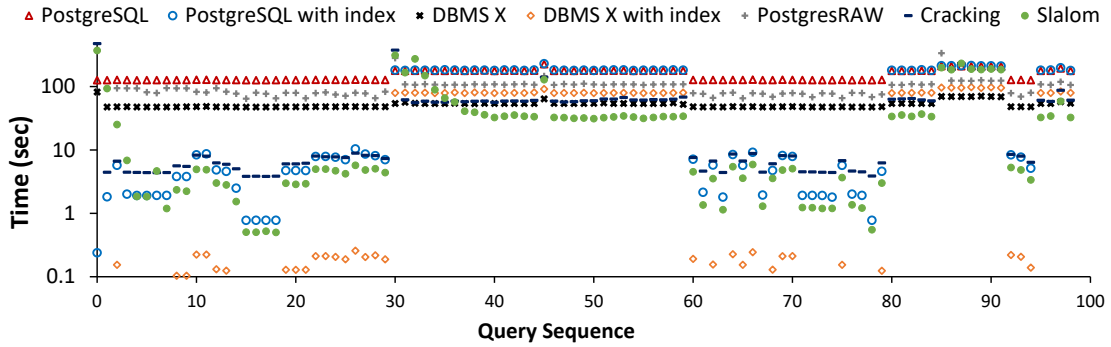

Figure 3.16 – Sequence of SHD analytics workload. Slalom offers consistently comparable performance to in-memory DBMS.

3.4.4 Slalom Over Real Data

In this experiment, we demonstrate how Slalom serves a real-life workload. We use a smart home dataset (SHD) taken from an electricity monitoring company. The data- set contains timestamped information about sensor measurements such as energy consumption and temperature, as well as a sensor ID for geographical tracking. The timestamps are in increasing order. The total size of the dataset is 55 GB in CSV format. We run a typical workload of an SHD analytics application. Initially, we ask a sequence of range queries with variable selectivity, filtering data based on the timestamp attribute (Q1-29). Subsequently, we ask a sequence of range queries which filter data based on energy consumption measurements to identify a possible failure in the system (Q30-59). We then ask iterations of queries that filter results based on the timestamp attribute (Q60-79, Q92-94), the energy consumption (Q80-84, Q95-100), and the sensor ID (Q85-91) respectively. Selectivity varies from 0.1% to 30%. Queries focusing on energy consumption are the least selective.

Figure 3.16 shows the response time of the different approaches for the SHD workload. All systems run from a hot state, with data resting in the OS caches. The indexed versions of PostgreSQL and DBMS-X build a B^+ tree on the timestamp attribute. The figure plots only query execution time and does not show the time for loading or indexing for PostgreSQL and DBMS-X. For other other systems, where building auxiliary structures takes place during query execution, execution time contains the total cost.

PostgreSQL and DBMS-X without indexes perform full table scans for each query. Q30-60 are more expensive because they are not selective. For queries filtering on the timestamp, indexed PostgreSQL exhibits 10× better performance than PostgreSQL full table scan. Similarly, indexed DBMS-X exhibits 17× better performance compared to DBMS-X full table scan. As the queries using the index become more selective, response time is reduced. For the queries that do not filter data based on the indexed field, the optimizer of DBMS-X chooses to use the index despite the predicate involving a different attribute. This choice leads to response time slower than the DBMS-X full scan.

PostgresRaw is slightly faster than PostgreSQL without indexes. The runtime of the first query that builds the auxiliary structures (cache, positional map) is 8× slower (374 sec) than the average query runtime. For the rest of the queries PostgresRaw behaves similar to PostgreSQL and performs a full table scan for each query.

After the first query, Slalom identifies that the values of the timestamp attribute are unique. Thus, it chooses to statically partition the data following the cost model for query-based partitioning (Section 3.3.1) and creates 1080 partitions. Slalom creates the logical partitions during the second query and calculates statistics for each partition. Thus, the performance of Slalom is similar to that of PostgresRaw for the first two queries. During the third query, Slalom takes advantage of the implicit clustering of the file to skip the majority of the partitions, and decides whether to build an index for each of the partitions. After Q5, when Slalom has stabilized partitions and already built a number of indexes over them, the performance is better than that of the indexed PostgreSQL variation.

Queries Q2-Q30 represent a best-case scenario for DBMS-X: data resides in memory and its single index can be used, therefore, DBMS-X is faster than Slalom. After Q29, when queries filter on a different attribute, the performance of Slalom becomes equal to that of PostgresRaw until Slalom builds indexes. Because the energy consumption attribute has multiple appearances of the same value, Slalom decided to use homogeneous partitioning. Q30 to Q59 are not selective, thus execution times increase for all systems.

Table 3.2 shows the costs for loading and indexing as well as the aggregate query costs for the same query workload of 100 queries, for all the systems. Due to the queries being non-selective, the indexed and non-indexed approaches of DBMS-X have similar performance, thus in total Slalom exploits its adaptive approach to offer competitive performance to the fully indexed competitors.

Summary. Slalom serves a real-world workload which involves fluctuations in the areas of interest, and queries of great variety in selectivity. Slalom serves the workload efficiently due to its low memory consumption and its adaptivity mechanisms, which gradually lower query response times despite workload shifts.

3.4.5 Slalom Handling File Updates

In this section, we demonstrate Slalom's update efficiency for append-like and in-place updates.

Append-like Updates

Slalom monitors changes in the queried files and dynamically adapts its data structures. In this experiment, we execute a sequence of 20 point queries following the template from Section 3.4 with selectivity 0.1%. Q1 to Q10 run on the original relation of 18 million tuples (22GB).

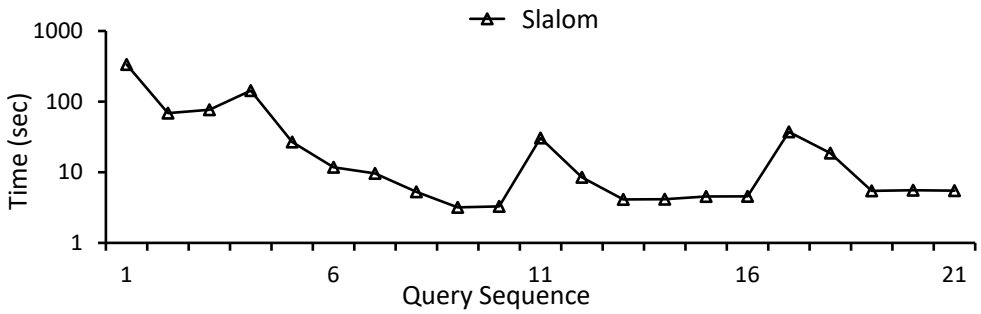

Figure 3.17 – Slalom executing workload with append-like updates.

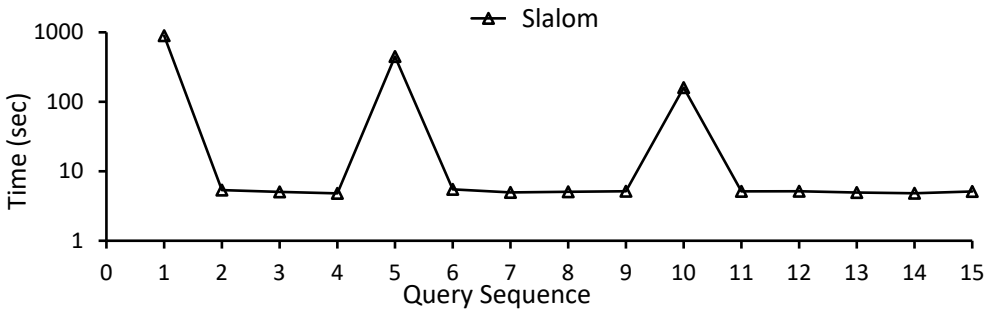

Figure 3.18 – Slalom executing workload with in-place updates.

Between queries Q10 and Q11 we append to the CSV dataset 6GB of additional uniformly distributed data. Slalom detects the change in the structure of the file and iteratively creates new logical partitions for the new tuples and creates Bloom filters and B^+ trees during Q11, Q12, and Q13. Between Q16 and Q17, we append again 6GB of data to the end of the CSV dataset. Slalom again dynamically partitions and builds indexes. Figure 3.17 shows the execution time for each of the queries in the sequence. Q11 and Q17 execute immediately after the appends, thus we see higher execution time because Slalom (i) accesses raw data, and (ii) builds auxiliary structures – positional maps and binary caches – over them. After this update-triggered spike in execution time, Slalom's partitioning and indexing schemes converge and the execution time becomes lower and stabilizes.

In-place Updates

We now show that Slalom handles in-place updates. We execute a sequence of 15 point queries following the template from Section 3.4 with selectivity 0.01%, run on a 25 million tuple relation (27GB). We query on a candidate key field to make Slalom use the query-based partitioning strategy and observe solely the effect of updates on a partition. To evaluate update efficiency, we develop a random update generator which updates fields and rows within a file in random places. Before query Q5, the update generator updates 8 random rows,

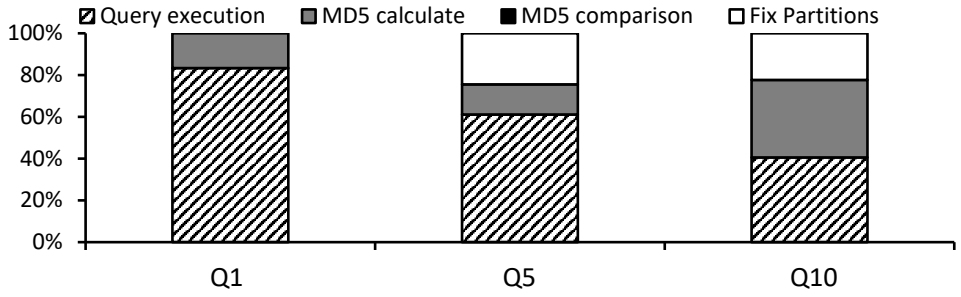

Figure 3.19 – Time break-down of query executiong with in-place updates.

and before query Q10, it updates 3 random rows. Figure 3.18 shows the execution time for each of the queries in the sequence. During Q1, Slalom creates 345 partitions and builds the positional map and indexes. During Q5 and Q10, the Update Monitor detects that the file has been updated. Slalom compares the state of all partitions to identify the updated partitions, performs the required corrections to the positional map, and re-builds the indexes. Figure 3.19 shows this process and presents the breakdown of query execution for Q1, Q5, and Q10. During Q1, along with query execution, Slalom calculates the MD5 codes for all partitions. The update before Q5 touched more partitions than the second update at Q10. Thus, Q5 has more partition data structures to fix. As the query execution progresses, the increasing number of partitions increases the number of checksum calculations.

Speed-up Checksum Calculation

This experiment examines the effect of using GPU and CRC accelerators for the calculation of the partition checksums. We execute 3 point queries following the template from Section 3.4 with selectivity 0.01%, over a 25 million tuple relation (27GB). To examine the efficiency of GPU and CRC checksum calculation we vary the number of partitions created by Slalom. The first query breaks the file into 100 equally-sized partitions, the second query into 1000 partitions, and the third into 10000 partitions. Before each query, we make a random update in the file to activate the re-calculation of checksums. Figure 3.20 shows the checksum calculation cost for the three queries using the three different appraches. When using the CPU (either the dedicated CRC instructions or MD5 calculation) the cost of calculation remains constant. On the other hand, when using the GPU, the checksum calculation is slower when the number of partitions is increasing. The best approach for calculating checksums is using the CRC. However, as CRC is able to compute checksums over input of 1024 byte blocks, it generates a large number of checksums for each partition. Thus, making checksum comparison more time-consuming.

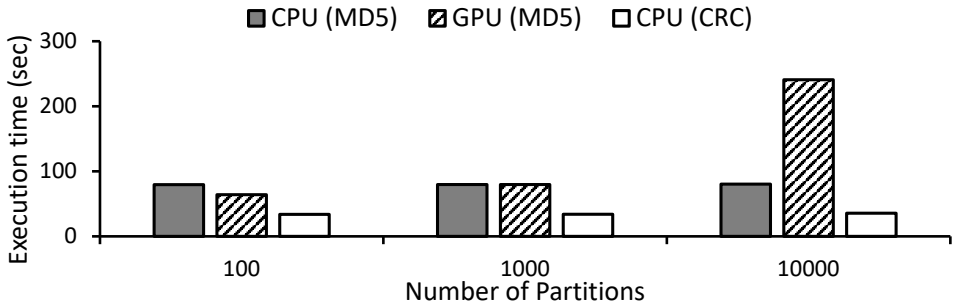

Figure 3.20 – Checksum calculation using different accelerators with different partition sizes.

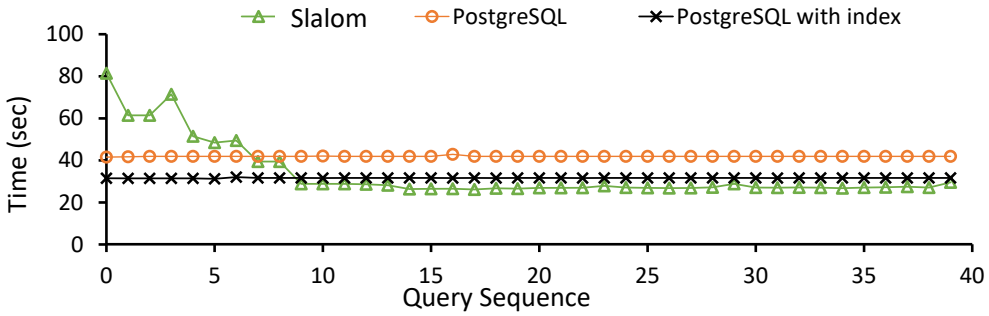

Figure 3.21 – Sequence of 40 queries over a binary file.

3.4.6 Additional Data Formats: Binary Data

This section shows that, besides CSV data, Slalom can also operate efficiently over binary datasets. Slalom employs the same techniques as when running over CSV files, with two exceptions. It tunes the cost model to reduce the access cost equations previously associated with text-based data accesses, and does not have to build a positional map. Figure 3.21 presents the performance comparison of Slalom and PostgreSQL with and without indexes. For this experiment we use a binary flat file with 100 million uniformly distributed tuples, each having 30 columns (12GB); we run range queries with selectivity 1%. For Slalom the initial data access is faster than that in the case of CSV data because (i) no parsing is involved and (ii) the binary representation is more compact than the CSV one. During the first 9 queries,

System	Loading	Index Build	Queries	Total
Slalom	0 sec	0 sec	1352 sec	1352 sec
PostgresSQL (with index)	325 sec	165 sec	1264 sec	1754 sec
PostgreSQL (no index)	325 sec	0 sec	1677 sec	2002 sec

Table 3.3 – Cost of each phase of the 40 query sequence on binary file.

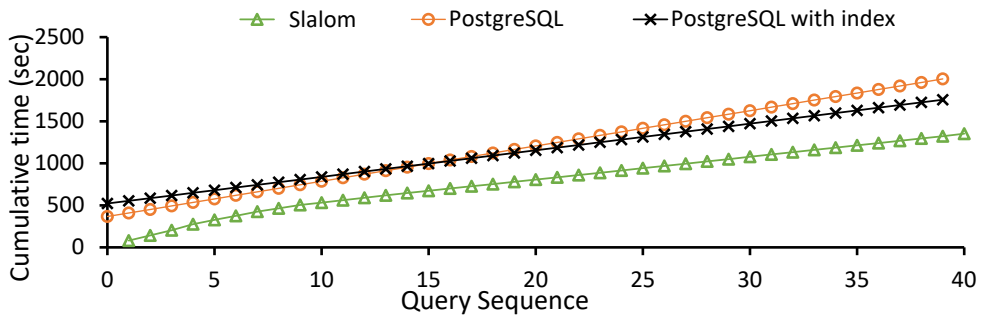

Figure 3.22 – Cumulative execution time of 40 queries over a binary file.

Slalom fine-tunes its partitioning. During Q3, multiple partitions happened to stabilize, thus triggering the construction of multiple indexes and leading to increased execution overhead. Both PostgreSQL configurations have stable execution times as the selectivity remains stable. Eventually, Slalom and indexed PostgreSQL converge and have similar performance. Figure 3.22 presents the cumulative execution time for loading, index building and query execution for the three systems over binary files. PostgreSQL using indexes requires more pre-processing time due to index building and it takes 13 queries to pay-off the cost of building the index. Slalom requires 7 queries to start outperforming PostgreSQL and after 10 queries it offers comparable performance to PostgreSQL with indexes. Table 3.3 presents separately the time required for loading, index building, and query execution for the three systems. The additional file adapters enable Slalom to efficiently and transparently operate on top of additional data formats.

3.5 Conclusion

In-situ data analysis over large and, crucially, growing data sets faces performance challenges as more queries are issued. State-of-the-art in-situ query execution reduces the data-to-insight time. However, as the number of issued queries is increasing and, more frequently, queries are changing access patterns (having variable selectivity, projectivity and are of interest in the dataset), in-situ query execution cumulative latency increases.

To address this, we bring the benefits of indexing to in-situ query processing. We present *Slalom*, a system that combines an in-situ query executor with an online partitioning and indexing tuner. Slalom takes into account user query patterns to reduce query time over raw data by partitioning raw data files *logically* and building for each partition lightweight *partition-specific* indexes when needed. The tuner further adapts its decisions on-the-fly to follow any workload changes and maintains a balance between the potential performance gains, the effort needed to construct an index, and the overall memory consumption of the indexes built.

4 Self-Tuning, Elastic and Online Approximate Query Processing

Current Approximate Query Processing (AQP) engines are far from silver-bullet solutions, as they adopt several static design decisions that target specific workloads and deployment scenarios. Offline AQP engines target deployments with large storage budget, and offer substantial performance improvement for predictable workloads, but fail when new query types appear, i.e., due to shifting user interests. To the other extreme, online AQP engines assume that query workloads are unpredictable, and therefore build all samples at query time, without reusing samples (or parts of them) across queries. Neither approach is capable of adapting query execution dynamically based on changes in the workload and underlying storage resources. As a result, current AQP engines miss out on opportunities for optimizing performance and cost. In this chapter, we present Taster, a self-tuning, elastic, online AQP engine that synergistically combines the benefits of online and offline AQP. Taster takes advantage of online AQP and performs online sampling by injecting samplers into the query plan while strategically materializing and reusing samples across queries. Taster continuously self tunes and adapts its materialized synopses based on changes in workload and underlying storage resources. We implement Taster over SparkSQL and use several industry-standard benchmarks to compare Taster with state-of-the-art online and offline AQP approaches. In doing so, we show that Taster can adapt to variations in workload and storage, and always converges to match, or outperform, the best performing AQP approach in all scenarios.

4.1 Introduction

In the past few years we have witnessed a renewed interest in approximate query processing due to two reasons. First, driven by the promise of big data analytics, enterprises started gathering data aggressively, collecting amounts that challenge state-of-the art exact analytics systems that require expensive, up-to-date hardware. Second, modern-day analytics use cases, like interactive data exploration, visual analytics, aggregate dashboards, and iterative machine learning workloads, are increasingly tolerant to imprecision. Approximate query processing (AQP) engines trade-off accuracy for better response time and lower resource usage by executing analytical queries over a sample of the data, and providing approximate results

within a few percents of the actual value.

State-of-the-art AQP engines are classified into two categories, depending on the assumptions they make about the query workload. *Offline AQP* engines (e.g. STRAT [30] and BlinkDB [6]) target applications where the query workload is known a priori, e.g., aggregate dashboards that compute summaries over a few fixed columns. Offline AQP engines analyse the expected workload to identify the optimal set of synopses (summaries of the data, such as samples, sketches, and histograms) that should be generated to provide fast responses to the queries at hand, subject to a predefined storage budget and error tolerance specification. Since this analysis is time-consuming, both due to the computational complexity of the analysis task, as well as the I/O overhead in generating the synopses, AQP engines perform the analysis offline each time the query workload or the storage budget changes.

While offline AQP engines substantially improve query execution time under predictable query workloads, their need for a priori knowledge of the queries makes them unsuitable for unpredictable workloads. Data exploration is one such example, where future queries are determined based on the results obtained from past queries. These workloads benefit from *online AQP* techniques, where approximation is introduced to query execution at runtime. State-of-the-art online AQP engines achieve this by introducing samplers during query execution. By reducing the input tuples, samplers improve performance of the operators higher in the query plan. In this way, online AQP techniques can boost unknown query workloads. However, query-time sampling is limited in the scope of a single query, as the generated samples are not constructed with the purpose of reuse across queries – they are specific to the query, and are not saved. Thus, online AQP engines offer substantially constrained performance gains compared to their offline counterparts for predictable workloads.

In summary, all state-of-the-art AQP engines force end-users to pick an extreme point in the generality–performance spectrum, as they make static, design-time decisions based on a fixed set of assumptions about the query workload and the available resources. However, workload in modern data analytics clusters is complex, far from homogeneous, and often contains a mix of queries that vary widely with respect to the degree of approximability [6]. Similarly, the available hardware resources are also non-static and time-varying. For instance, an administrator might elastically provision storage space for storing synopses based on the expected system load. Hence, in the ideal case, an AQP engine should be *self-tuning* and *adaptive*. It should automatically pick the right point in the design spectrum based on the workload, and adapt its decision on-the-fly with each change in workload or storage capacity.

In this work, we present Taster, a self-tuning, elastic, online AQP engine that can adapt dynamically to variations in workload and storage. Taster inherits ideas from (adaptive) database systems, such as intermediate result materialization, query subsumption, materialized view tuning and index tuning, and adapts these in the context of AQP, enabling a combination and extension of the benefits of both offline and online approximation engines. Central to the approximation approach of Taster is that it operates at the level of query planning, i.e.,

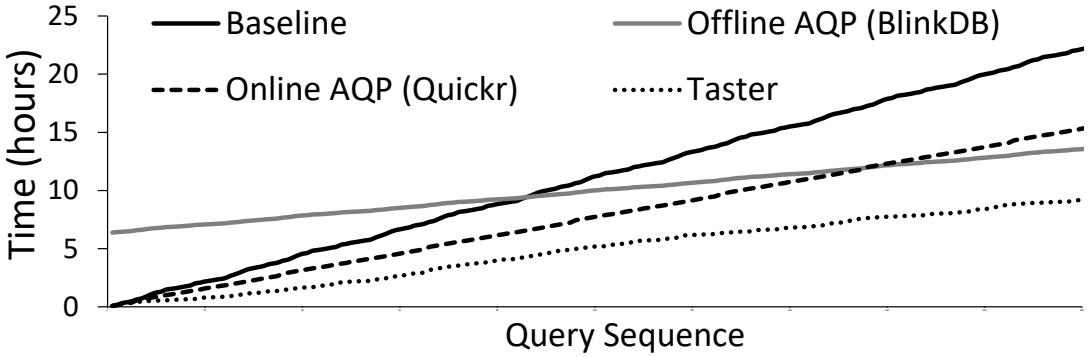

Figure 4.1 – Cumulative time-to-insight for various approaches in an exploratory data analysis usecase.

subplans are promoted as first-class citizens. This enables Taster to materialize and save synopses on intermediary results, e.g., a frequently-executed join, and to use these synopses as auxiliary access paths, for the purpose of query planning of future queries. Furthermore, Taster builds on an extensive and extensible arsenal of approximation techniques, combining different samplers and sketches in order to cover different families of queries.

Taster combines and extends the benefits of both offline and online approximation engines. First, by injecting approximation operators in the query plan, Taster offers high generality similar to online AQP engines as it can support both precise and approximate queries in a single engine. Taster extends prior work by showing that the technique of injecting sampling operators can also be generalized to sketch-based approximation solutions. Second, by materializing and reusing samples, Taster provides performance on-par with offline AQP engines under predictable workloads. Taster extends prior work by supporting sample and sketch materialization at intermediate stages in the query plan instead of restricted base table materialization. Third, by using online data structures to track subplan similarities across queries, and a cost:utility greedy algorithm to determine the right set of synopses to maintain, Taster can adapt on-the-fly to both changes in workload and available storage.

Thus, an administrator can fully exploit cloud storage elasticity by dynamically changing the budget used for sample storage to match expected workload demand without having to take the system offline for reconfiguration.

Example scenario. To illustrate the utility of Taster, let us consider an example use case. Visual analytics is a core process in data science, facilitating extraction of useful insights out of big data [57, 79]. A data scientist typically starts by running simple exploratory queries over the data and visualizing the results, formulating and validating hypotheses. Queries are not known a priori, since each query typically depends on the results of the previous queries. For example, the results of one query may hint the user to zoom in, or to analyze further a particular region of the data as the next query. In this case, offline AQP engines

cannot be used, since they require a priori knowledge of the query load in order to prepare the synopses. Even if this information is somehow provided by an oracle, then offline AQP engines will require a potentially huge preparation overhead for constructing all synopses before any query can be approximated (cf., series Offline AQP (BlinkDB), Figure 4.1 – the details of the experiment will be discussed at Section 4.5). On the other hand, online AQP engines such as Quickr can be used, and they offer a substantial performance improvement compared to not using approximation at all (cf., series Online AQP (Quickr), Figure 4.1). However, online AQP engines do not support reusability of approximations across queries (e.g., if two queries have an overlapping sub-plan). The ideal situation (cf., series Taster, Figure 4.1) is to start building the synopses as byproducts of the queries, and save these synopses such that they can be reused in future queries. A synopsis can be built on a base relation (a table), or even on intermediary result, e.g., the results of a join, or even the results of a filter. Since saving of a synopsis carries a cost, the decision as to which synopsis to keep is taken by the query engine, considering the utility of each synopsis and the frequency of use at the recent queries. Furthermore, the storage budget for synopses can be increased or reduced at will by the administrator – or even automatically using simple threshold rules – in order to anticipate an increase in data, query load (number of users), and available hardware.

Contributions. This chapter makes the following contributions:

- We present an online adaptive approximate query processing approach that enables materialization of synopses during query execution, and their reuse across queries. Our approach uses synopses for summarizing both base tables and intermediary results of query subplans. As a results, the online adaptive AQP approach removes the requirements for preprocessing, improves query performance and reduces storage requirements.
- We show how to integrate synopses as first-class citizens in query planning, which leads to better plans and improved performance.
- We present an online algorithm determining the optimal set of synopses to maintain by using a utility metric that captures the performance benefit of a synopsis.
- We present other possible optimizations assuming additional knowledge of user's intentions (e.g., some frequent queries, on which attributes, and on which files). We sketch the space of possible optimizations in the presence of additional user hints, and demonstrate how to integrate these hints by pre-constructing some samples.
- We integrate our techniques into SparkSQL and create our prototype system named Taster. We compare Taster to vanilla SparkSQL, a representative offline AQP approach called BlinkDB, and an online approximation approach called Quickr. Our experiments with industry-standard benchmarks demonstrate that Taster offers substantially improved performance compared to online AQP engines (2.9×), and comparable performance to offline AQP engines without requiring the excessive sample pre-generation

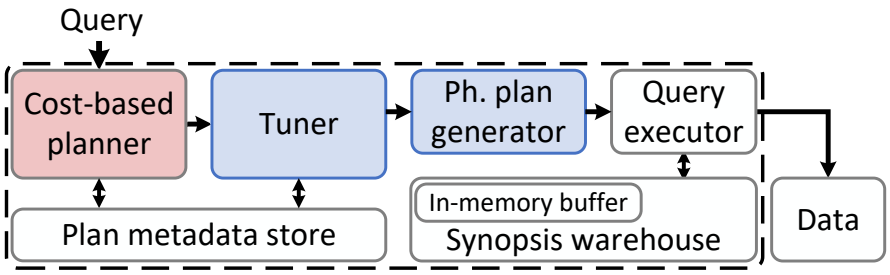

Figure 4.2 – The overview of Taster.

cost. Speed-up compared to the baseline is over 3× on average (20× when additional hints are provided, reaching to 30× for some queries).

The rest of this chapter is organized as follows. Section 4.2 discusses the architecture of Taster, along with an example of its execution workflow. Section 4.3 details on the query planning process, whereas Section 4.4 discusses the self-tuning nature of the system. Section 4.5 presents a thorough experimental evaluation. We conclude in Section 4.6.

4.2 Architecture of Taster

Figure 4.2 presents Taster's high-level architecture. Taster is implemented over SparkSQL and extends Apache Catalyst query optimizer and SparkSQL query engine with online approximation techniques, combined with synopsis materialization and self-tuning. The techniques presented are not limited to SparkSQL, and are applicable to any query processing system – even centralized ones. In the following we present a high-level overview of the core concepts of Taster.

Synopses and synopsis warehouse. Taster uses a set of automatically-constructed and tuned synopses to summarize both the raw data (the base relations) and intermediary results of subplans (e.g., join results). Currently, it exploits two types of synopses, samples and sketches, each being appropriate for answering different query families. All synopses are constructed as byproducts of query answering, and are saved in the synopsis warehouse, in HDFS. Along with synopses, Taster stores statistics of the dataset (distribution of values, number of distinct values), which are calculated on-the-fly during the first access to any table. To control monetary cost, the synopsis warehouse is subject to space quota, which is set at initialization and can also be modified at runtime from the administrator. More details for the process of selecting synopses for the synopsis warehouse will be presented in Section 4.4.

Synopsis buffer. The plan chosen for execution may require generation of a new synopsis (i.e., if the synopsis is not already in the synopsis warehouse). Generation of a new synopsis on-the-fly may still be beneficial for the query at hand, in order to reduce CPU usage of

operators higher in the plan. In this case, the new synopsis will be temporarily stored in the *synopsis buffer* – a fixed-size buffer implemented as a sequence of in-memory RDDs in Spark. The buffer offers two main benefits: (a) it serves as a fast main-memory cache, which offers significant boost for workloads exhibiting temporal locality, and, (b) it decouples the decision of writing the synopsis in the HDFS-based synopsis warehouse – an I/O expensive operation – with the process of query answering which needs to be executed with a very small latency. When the buffer is full, the tuner decides which synopses should be permanently stored in the synopsis warehouse (cf. Section 4.4).

Cost-based planner. Taster's query engine decides automatically on the exploitation of supported synopses to speed-up user queries. This automation relies on a *cost-based planner*, which is currently built into the Catalyst optimizer. Upon receiving the query, the planner generates a set of approximate execution plans. These plans utilize synopses that may, or may not yet exist, and they all satisfy the approximation requirements of the query. The next step is to estimate the cost of each plan and the performance gain by the use of synopses, compared to the best plan without synopses that will return exact answers. The plans and their costs are then passed to the tuner, for further optimizations and the final execution. The cost-based planner is discussed in Section 4.3.

Tuner. The primary purpose of the tuner is to choose the best plan out of the ones proposed by the planner. However, when ranking the plans, the tuner focuses on maximizing long-term throughput, i.e., over the future workload, as opposed to minimizing the cost for the query at hand. This *holistic optimization* translates to decisions in two levels: (a) promoting the plans that generate reusable synopses, pertinent to many different queries, and, (b) deciding which of the generated synopses will be stored in the synopsis warehouse, and which will be deleted, to satisfy the space quota. Tuning involves two major challenges: (a) holistic optimization can be CPU-intensive, and (b) the future queries, over which the tuner needs to optimize, are of course not yet known. We explain how these issues are addressed in Section 4.4.

Physical plan generator. The plan chosen by the tuner is subsequently passed to the physical plan generator, for extraction of the physical plan and execution over Spark. The physical plan generator is now implemented within the tuner to avoid additional synchronization overhead. Fault tolerance, distribution, partitioning-related details, and the actual task execution are handled transparently by Spark.

Metadata store. Effectiveness of both the planner and tuner depends on the existence of metadata that characterizes the past workload and the synopses that could speed-up this workload. The metadata store is a main-memory, *synopses-centric* metadata repository that keeps rich statistics about the properties, impact, and popularity of each synopsis. In particular, the store keeps details for all synopses contained in all plans generated by the planner – even the ones that are not chosen for execution. These details include: (a) the logical definition of the synopsis (the logical subplan whose results are summarized by this synopsis), (b) stratification

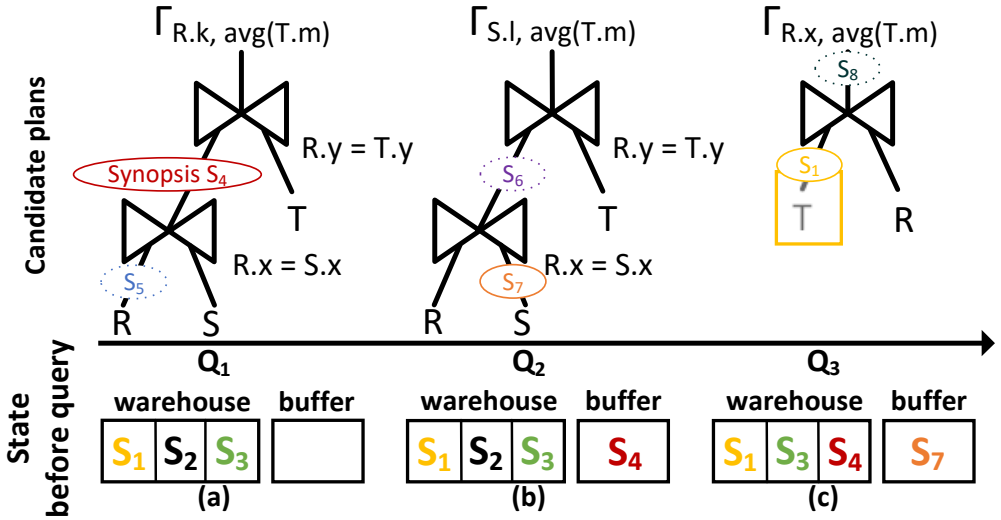

Figure 4.3 – Example execution of Taster. The dotted ellipses depict prospective synopses, the normal-line ellipses depict chosen synopses. Yellow boxes are used to denote reuse of existing synopses. $\Gamma_{group,aggr}$ denotes aggregation operator with $group$ grouping attributes and $aggr$ aggregation function.

and accuracy requirements of the synopsis, (c) whether the synopsis is saved in the synopsis warehouse or not, and, (d) the list of recent queries that could utilize this synopsis to improve performance, their estimated cost when this synopsis exists, and their cost if an exact query plan (without synopses) would be chosen instead. The purpose of this metadata is twofold: (a) to assist the planner to estimate the cost of each candidate plan (cf. Section 4.3.1), and (b) to enable the tuner to decide which synopses will maximize throughput, i.e., because they will improve many different subplans (cf. Section 4.4).

Example. Figure 4.3 presents an overview of Taster running three queries over three relations R, S, T. For simplicity, we assume that the synopsis buffer fits one synopsis, and the warehouse fits three synopses. [1] Just before arrival of Q_1, the synopsis warehouse already contains synopses S_1, S_2, and S_3. S_1 is a sample of relation T. Synopses S_2 and S_3 refer to another table W, not relevant to the three queries.

During Q_1, the planner proposes two candidate plans (cf., Fig. 4.3a). The first one contains synopsis S_4, which summarizes $R \bowtie S$, and the second contains synopsis S_5 of R. Notice that neither of the two synopses exist. The two plans are costed, and the metadata store is updated with the corresponding properties of S_4 and S_5. Then, the plans are sent to the tuner. The tuner identifies the best plan (in this case, the one with S_4), and sends it for execution. During execution, S_4 is generated and saved in the in-memory synopsis buffer. When Q_2 arrives, the planner identifies two candidate plans (cf., Fig. 4.3b), which rely on the nonexistent synopses

[1] This is only for illustration purposes. Since synopses have different sizes, quotas are determined in GB, and not in number of synopses.

S_6 and S_7 respectively (synopses S_1 and S_4 cannot be used because of different grouping attributes). Again, the planner updates the metadata store with the corresponding properties of the two candidate synopses, and the plans are sent to the tuner. Now, as the synopsis buffer is full, the tuner first needs to free up space, so that either of the candidate synopses can be generated. The synopsis warehouse is also full. By estimating the long-term benefit of each synopsis, the tuner decides to keep S_1, S_3, and S_4 in the warehouse, and to execute the plan that requires S_5. The plan is executed, and S_7 is stored in the synopsis buffer. During Q_3, the planner proposes two plans (cf., Fig. 4.3c), the first replacing the scanning of relation T with S_1 which is already saved in the warehouse (the yellow box), and the second utilizing a non-existent synopsis S_8. The plans are sent to the tuner, where the first one is chosen and sent for execution.

Supported Queries. Taster accepts and answers all SQL queries supported by Spark SQL. Similar to prior work, e.g., [6, 70], it improves performance for queries containing aggregates (e.g., COUNT, AVG, SUM). The query format for approximate queries follows the standard syntax: "ERROR WITHIN x% AT CONFIDENCE y%", which corresponds to aggregate results with relative error of at most x% at a y% confidence level. Taster adapts the query plan accordingly such that the accuracy guarantees are satisfied and all groups are included in the results, e.g., when a group-by is requested.

4.3 Query Planning with Synopses

Taster automatically decides which synopses to create, store, and use for answering each query. Synopses are used for summarizing both raw data (base relations) and query subplans (e.g., the results of an aggregator over a join). Due to their small size compared to the original data, synopses improve both computational complexity and I/O cost during query processing. All synopses are created on-the-fly, as byproducts of query answering, thereby inducing no additional I/O.

Synopses in Taster are promoted to first-class citizens: they are included as approximate operators in the logical query plans, costed as all other logical operators, and transformed to fully pipelined and distributable code during the physical plan generation. This enables the planner to produce more efficient plans, and the tuner to promote reusability of synopses by matching synopses across different queries. In the remainder of this section we will explain how the Taster planner integrates synopses into planning. The discussion explains the plan generation process, how synopses are configured to satisfy the query's accuracy requirements, and how they are matched to existing synopses from the synopses warehouse.

4.3.1 Query Planning

The planner generates auxiliary logical plans, replacing the aggregator operators with approximate aggregators (whenever these are beneficial for performance), costing the plans, and,

passing them to the tuner for further optimizations. In the following, we describe this process in detail.

Upon receiving query q_i, the planner generates candidate logical plans $\mathbb{P}(q_i) = \{p_1, p_2, \ldots\}$, which integrate synopses. The key observation to limit the search space is that prospective synopses are used for approximating aggregators and joins. Focusing on aggregations, the planner first identifies all query subplans rooted on (partial/eager) aggregators. For each, it injects a generic synopsis operator just below the aggregator operator, and modifies the aggregator to account for the synopsis (e.g., a SUM over a sample would require scaling to account for the full dataset). The synopsis operator represents the potential to efficiently approximate the underlying subplan by the use of a (possibly not yet existent) synopsis. Subsequently, Taster tweaks the query plan to achieve two goals, (i) maximize the re-use of existing synopses and (ii) satisfy the user's accuracy requirements. All resulting plans are annotated with cost estimates based on their expected I/O, and analyzed to extract all synopses, along with the subplans they summarize. The collected data is used to update the metadata store with the appearances of these synopses. Following, all plans are passed to the tuner for further optimizations.

The above process entails several challenges. First, the process of generating candidate plans is different compared to traditional planners. Unlike traditional query planning, the planner now also needs to take into account the required approximation guarantees and stratification requirements while constructing the plans. Furthermore, when pushing down a synopsis in the plan, the synopsis, as well as its corresponding approximate aggregation operator, may require modifications. Second, the approximate aggregators in the plan need to be configured. This boils down to choosing between the supported types of sampling and sketches, and configuring the selected synopses (e.g., for uniform sampling, setting the sampling probability). Third, the candidate synopses contained in the plan need to be mapped to existing synopses (if any), so that the planner can replace the subplan with the synopses, and estimate the execution cost. In the following we describe how the planner handles these three challenges.

Generating the candidate plans. The planner generates the first set of plans by injecting synopsis operators below the aggregations. Particularly, given aggregation operator $\Gamma_{\mathcal{G},AGG(\mathcal{A})}(c)$, which computes aggregation function AGG over the data produced by operator c (the child operator in the logical plan) by grouping over attributes $\mathcal{G}$, the synopsis operator $\Gamma^{\mathsf{S}}_{state}$ is injected and the aggregation operator is updated (now denoted as $\Gamma'_{\mathcal{G},AGG(\mathcal{A})}(\Gamma^{\mathsf{S}}_{G}(c))$) to use the synopsis as input. Subsequently, Taster starts pushing the synopses down in the plan, closer to the raw data, as an effort to enable executing the plan with existing synopses, or to generate more re-usable synopses. For these, it relies on the push-down rules for synopses introduced in [70], and adapted to enable sketch synopses. Briefly, whenever Taster pushes a synopsis operator under a filter σ_p, it needs to account for two possibilities. If the distribution of values of predicate p is uniform, the new operator is moved under the filter unaltered, since a uniform sample over that attribute will not reduce the number of groups appearing

in the final result [84]. However, if the distribution of the values of p is skewed (some groups appear infrequently), Taster needs to stratify the underlying output on p. Thus, Taster adds the attributes appearing in p which follow a skewed distribution into the stratification set.

Considering pushing synopses under the joins, given a join $R \bowtie_{jp} S$ with join predicates jp, the planner pushes the synopsis below the join, to the side of the join on which the aggregation takes place (say, the side of R), and modifies the stratification attributes of the synopsis to include the attributes from jp that are contained in R (i.e., $\Gamma^{S}_{(\mathscr{A} \cup jp) \cap R}(R) \bowtie_{jp} S$). Finally, if the join predicate is not a grouping attribute, Taster introduces a partial aggregation after the join.

The above push-down process guarantees that (i) the generated physical query plan will gather sufficient samples from each of the groups to satisfy user's accuracy requirements, and (ii) the overall sampling process overhead will not exceed the performance gains. We discuss how result accuracy is estimated efficiently and reused across different queries in Section 4.3.2. In terms of implementation, the push-down strategies are implemented as rules in the Catalyst optimizer, and are executed at every query. Since Catalyst default implementation returns only a single plan at the end, we intervene the planning process in order to store all intermediate plans.

Choosing and configuring the synopses. The synopsis operators contained in the logical plans up to now were parameterized with stratification and accuracy requirements, but omitted configuration details, e.g., which synopsis to use, and how to configure it for satisfying user's accuracy requirements.

Due to the immense ratio of performance gain to storage requirement of sketches, Taster prioritizes the use of sketch-join when appropriate: Let R and T be two relations joined over attributes jp and subsequently passed through aggregator $\Gamma_{grp,agg}$, with grp being the grouping attributes and agg the attributes taking part in the aggregation. With $attrs(R)$ we denote the attributes of R which are given as input to the join. Sketch-join can boost join queries with aggregates, when the projected attributes from one side of the join are either join attributes, or they are used in the aggregate function. Formally, the following requirements must be satisfied:

- $attrs(T) - jp = agg$
- $attrs(T) \cap grp = \emptyset$ OR $attrs(T) \cap grp = attrs(T) \cap jp$

Then, the synopsis operator injected between the aggregation and the join can be pushed under the join operator, and transformed into a sketch-join operator.

When sketch-join is not applicable, Taster falls back to sampling. In this case, the planner needs to decide which sampling strategy will be used. A key input for this decision is the cardinality estimates per relational expression, and the number of distinct values in each column (both

these statistics are computed during the first access to the table). In particular, Taster checks (i) if the set of stratified attributes C is empty, and, (ii) if some sampling probability $p \leq 0.1$ can ensure that, each distinct value of the columns in C receives at least k rows w.h.p.. If both these checks are true, the sampler is implemented using the uniform sampler. Otherwise, if $C \neq \emptyset$, Taster chooses a distinct sampler. Finally, Taster generates a plan without samplers if stratification and accuracy requirements are so restrictive that they cannot be satisfied with a reasonable sampling probability.

Matching subplans to materialized synopses. Costing of the logical plans requires efficiently matching the synopses contained in the query's logical plans to the synopses stored in the synopsis warehouse and buffer. This matching is enabled through the metadata store.

Particularly, each synopsis (candidate or materialized) corresponds to a unique logical subplan – the one of which the results it summarizes. Therefore, the subplans for the query at hand are compared to the subplans of the synopses contained in the metadata store. We say that a query subplan matches a synopsis when: (i) the accuracy guarantees of the synopsis satisfy the query requirements, and (ii) the synopsis subplan subsumes the query subplan. For the latter, Taster ensures that the query subplan is covered by the synopsis regarding join and filtering predicates as well as the projected columns. Particularly, Taster compares the input relations, the join and filtering predicates as well as the output attribute set. The synopsis subplan must have identical join predicates, its filtering predicates must be weaker than, or equal to the filtering predicates of the query, and its output attributes must be a superset of the corresponding parameters of the query subplan [50]. Some mismatches are addressed by adding filtering and projection operators directly above the query subplan, to remove extraneous tuples and attributes. Considering accuracy, a synopsis is a candidate for a subplan if (i) the set of stratification attributes of the stored synopsis is a superset of the stratification attributes of the subplan, and (ii) the aggregation function and the aggregate columns are identical to those of the synopsis and the accuracy requirement of the query generating the synopsis is equal or weaker than of the current query. By ensuring the former, Taster guarantees group coverage i.e., Taster results will contain all groups, whereas the latter ensures that the aggregates will have constrained error [6]. For example Q1: "SELECT $dept, AVG(salary)$ FROM $Employees$ GROUP BY $dept$" will generate a sample over $Employees$ stratified on $dept$. Subsequent query Q2: "SELECT $dept, AVG(salary)$ FROM $Employees$ WHERE $gender$ = 'male' GROUP BY $dept$" will be able to use the previous sample, since, the created sample is more general and Taster can put an additional filter in the query plan. However to use this sample, salaries should be uniformly distributed, irrespective of gender.

Subplan matching is expensive. Therefore, Taster utilizes an index to speed-up this process. Specifically, all candidate synopses contained in the metadata store are indexed using their base relations as the key. In the case of joins, the join attribute(s) are also included in the key. This index, although simple, effectively limits the search space and the lookup time to find suitable synopses for each subplan.

4.3.2 Accuracy guarantees

While generating and exploring the potential plans, the planner needs to ensure that the user's accuracy requirements are satisfied. For this, Taster relies on previous analytical results [38,70], which we outline below.

When using sampling, Taster uses the Horvitz-Thompson (HT) estimator [84] to calculate unbiased estimators of the true aggregate values. Confidence intervals are computed using the CLT. Due to the distance of the samplers to the aggregation operators, we use the notion of dominance between query expressions as defined in Quickr [69], which ensures that plans resulting from transformation rules used by the optimizer have no worse variance of estimators and no higher probability of missing groups than the plan with only one sampler before the aggregation operator. In terms of implementation, a naive way to compute the HT estimator squared error requires a self-join and can take quadratic time since it checks all pairs of tuples in the sample [84]. However, for stratified and uniform sampling, Taster calculates the error in a single pass by utilizing the observation of [70] that to compute the standard error for each group we only need to take into account the tuples with the same stratification key (resp. grouping key). Therefore, we estimate the expected error for each group by building a distributed hash table, using as a key the values of the stratification (resp. grouping) attribute, as as value the running estimated error for that group and the corresponding list of sampled tuples. For every sampled tuple, Taster updates the error of that tuple's group by using the HT estimator error formula, leading to a single-pass, linear complexity algorithm.

CM-sketches offer error guarantees relative to the L1 norm of the summarized relation [38]. Particularly, let $f(x)$ denote the real frequency of key x, and $\widehat{f}(x)$ the frequency estimated from the sketch. Then, the sketch is configured such that $\widehat{f}(x) - f(x) < \epsilon N$ w.h.p., where N represents the L1 norm of the frequencies for all keys.

4.4 Continuous synopsis tuning

Taster's self-tuning nature and ability to adapt to shifting user interests stems from a lightweight synopsis tuner. The tuner is invoked just after the planner, and has a goal to select the candidate plan that will *maximize the throughput over a window of the next w queries* (we will discuss about the value of w later). That is, in contrast to the planner which generates plans with a short term outlook (per-query performance), the tuner looks into overlaps between queries and query subplans in order to increase the long-term performance. The tuner's decisions are driven by a cost:utility model, which leads to a formalization of the task as an optimization challenge. Notice that the decisions made by the tuner affect solely query performance, and not the required accuracy. Even though the tuner has the final decision on which synopsis to build, the considered synopses are proposed by the planner, and thus satisfy user's accuracy requirements (cf., Section 4.3.1). Intuitively, if a synopsis is expected to be used across many queries to approximate different query subplans, and this synopsis offers significant boost, then the subplan that will generate this synopsis should be chosen.

The cost:utility model. Tuning is an iterative process. At every invocation, the tuner is presented with a set of candidate plans for query q, denoted with $\mathbb{P}(q) = \{p_1, p_2, \ldots\}$, and needs to choose one for execution in order to maximize throughput. Intuitively, the tuner will solve two problems concurrently: (a) select the best plan and corresponding synopses for answering the query, and (b) choose the best set of synopses to keep, which will speed-up Taster over a horizon of the next w queries, denoted with $\mathbb{Q}_i^+$, i.e., $\mathbb{Q}_i^+ = \{q_i, q_{i+1}, \ldots, q_{i+w-1}\}$.

It is useful to define the synopsis gain metric, i.e., how much does each set of synopses $\mathbb{S}$ contribute to the performance of each query. Formally, $gain(q,\mathbb{S}) = cost(q,\emptyset) - cost(q,\mathbb{S})$, where $cost(q,\mathbb{S})$ denotes the *minimum cost of any plan* in $\mathbb{P}(q)$ for answering q, given only the synopses in $\mathbb{S}$. In the case of $\mathbb{S} = \emptyset$, this will be the cost of the most efficient plan that does not utilize synopses and returns the exact answers. For a given $\mathbb{Q}_i^+$ we maximize the query throughput by minimizing the total cost of these queries, i.e., $\texttt{minimize} \sum_{q\in\mathbb{Q}_i^+} cost(q,\mathbb{S})$, or equivalently, by maximizing their corresponding gain: $\texttt{maximize} \sum_{q\in\mathbb{Q}_i^+} gain(q,\mathbb{S})$. For convenience, we slightly overload the notation by using $gain(\mathbb{Q}_i^+,\mathbb{S})$ to denote the gain over all queries using synopses in $\mathbb{S}$. Notice that the problem contains two variables. The first one, which is latent, is the set of plans $\mathbb{P}(q)$ for each query $q \in \mathbb{Q}_i^+$. The second is the set of synopses $\mathbb{S}$. Formally, the optimization problem is as follows:

$$\begin{aligned} \underset{\mathbb{S}}{\texttt{maximize}} \quad & gain(\mathbb{Q}_i^+,\mathbb{S}) \\ \text{subject to} \quad & \sum_{s\in\mathbb{S}} |s| \leq \texttt{maxSpace} \end{aligned}$$

where `maxSpace` denotes the space quota for synopses, and $\mathbb{S}$ denotes the set of synopses that will maximize the objective function. Therefore, the tuner needs to select the set of plans (one per query) and synopses that will maximize the total gain.

Even though the problem is well-defined, it involves two challenges. First, it turns out that the problem can be reduced to a variant of the NP-hard knapsack constraint problem. This happens because of correlations between synopses, i.e., each synopsis can be used for answering more than one queries, and some queries are answered by more than one synopses. Therefore, we cannot hope for a tractable exact solution. Luckily, we can approximate the solution within a constant factor by noticing that the objective function is a monotone submodular function, i.e., the gain provided by each single synopsis is only reduced as the set of synopses in $\mathbb{S}$ increases. For this special case, there exist several efficient approximation algorithms. We employ the efficient greedy algorithm of [81], which guarantees that the gain of the constructed set will be within a factor $(1-1/e)/2$ of the maximum gain. In a nutshell, the algorithm builds $\mathbb{S}$ gradually by starting from an empty set and adding synopses one-by-one until the quota is filled. At each step, synopses are chosen based on their marginal gain, i.e., how much is the additional gain each synopsis brings when added in $\mathbb{S}$. After $\mathbb{S}$ is created, the tuner checks all synopses that are already stored in the synopsis buffer and warehouse, and updates them accordingly: all synopses not contained in the newly-computed $\mathbb{S}$ are deleted.

The second challenge concerns the definition of the tuner's horizon, $\mathbb{Q}_i^+$. In practice, we

cannot expect to know the queries contained in $\mathbb{Q}_i^+$ during the tuning. We therefore employ the standard assumption that recent queries are a good representation of the following queries [26]. For this, we keep track of the last w queries, denoted as $\mathbb{Q}_i^- = \{q_{i-w+1}, q_{i-w+2}, \ldots q_i\}$, and use their proposed plans to estimate $gain(\mathbb{Q}_i^+, \mathbb{S})$.

Storage elasticity. This cost:utility model is also used for adapting to the available storage budget. Taster's administrator can modify the space quota of the synopses warehouse online. This action will automatically invoke the tuner to re-evaluate all synopses, and decide which ones need to be discarded, or created at future queries.

Physical plan generation. The above algorithm will choose both set $\mathbb{S}$, and the plan that minimizes the cost for q. This plan is then used for generating the physical plan. If the plan refers to creation of a new synopsis, then this step is injected in the physical plan as a new operator. The new synopsis is then stored in the in-memory synopsis buffer. In this case, the tuner has already freed up the required space in the buffer, during the tuning phase.

Computational overhead of the tuner. The cost estimates for each subplan (with and without each synopsis) are already computed by the planner and stored in the metadata store, i.e., they do not need to be recomputed from the tuner. The tuner also knows which of the synopses are already stored in the synopsis warehouse or the synopsis buffer, in order to account for the need to create synopses that do not yet exist. Therefore, computation of marginal gain per synopsis is very efficient. In practice, our single-threaded/centralized implementation of the tuner takes ~ 2 seconds per query.

Adapting the tuner's horizon length. To predict usefulness of each synopsis, Taster uses a sliding window of the previous w queries as a good approximation of the next, unseen w queries. The best value for w depends on the task at hand – data exploration, verification of hypotheses, finding outliers, etc. – which determines the *repetitiveness* in the query workload. Therefore, Taster dynamically adapts w.

Initially, w is set to a small value. The tuner also identifies (without building) the set of best synopses using a slightly larger and a slightly smaller w value, i.e., $w^+ = \lceil (1+\alpha) \times w \rceil$ and $w^- = \lfloor (1-\alpha) \times w \rfloor$, with $\alpha \in (0,1)$. At the next invocation, the tuner examines which of w^-, w, or w^+ would minimize execution time for the queries that arrived since the last invokation, and sets w to that value for the next tuning round. Since all necessary statistics for estimating execution time are already contained in the metadata store, this computation is very efficient.

Our experimental results signify the need to dynamically adapt w. In our tests, we start with default values $w = 10$, and $\alpha = 0.25$. The results show that, for the tested query workloads, the optimal w varies between 12 and 17. Compared to a fixed w, adaptive configuration shows performance improvement that exceeds 1.5×. A too large or too small value of w annihilates the predictive nature of the tuner, leading to bad choice of synopses. Value of α is also important on the adaptation speed, and part of our current work is to vary α.

User hints. Our discussion up to now assumed that the user is not required to (and, in most cases, cannot) offer hints/advice to the system. This is the typical case in many data science and data exploration scenarios, where the query load is unpredictable – hence the importance of the online tuner. However, several past works frequently required that the user provides different types of hints for the optimizer. This information includes, e.g., the whole query workload [6], or the synopses to be constructed, such that they can be build in a pre-processing step [96]. The natural question that arises is: how can Taster utilize such additional knowledge and hints?

A priori knowledge of the full query workload can be utilized from Taster, for accurate computation of the gain of each synopsis – since the full $\mathbb{Q}_i^+$ will be known at every invocation of the tuner, we do not need to revert to the past queries $\mathbb{Q}_i^-$ in order to estimate $gain(\mathbb{Q}_i^+, \mathbb{S})$. The user can also request some synopses to be pre-built offline, and pinned in the synopsis warehouse. In this case, Taster will generate these synopses off-line, and the tuner will never delete them. Still, tuner will keep optimizing the use of the remaining available space, filling it with synopses according to the observed queries. As we show experimentally, pre-computed synopses can lead to significant speed-up (up to 20× compared to baseline), since the synopsis generation time will not be included in the query execution time.

4.5 Evaluation

We compare Taster against three state-of-the-art systems: Quickr [70], BlinkDB [6][2], and vanilla SparkSQL which we refer to as *Baseline*. We compare the systems using industry standard benchmarks and a micro-benchmark. Specifically, we use TPC-H with scale factor 300 (300GB before compression) along with the TPC-H queries[3], and TPC-DS with scale factor 200 (200GB before compression) along with a set of 20 TPC-DS queries. To examine suitability of Taster under various workloads we also use a synthetic benchmark of an online grocery store (*instacart*) [1], scaled 100× (~ 120GB before compression). The query templates used for the instacart benchmark are shown in Table 4.1. All datasets were stored in the Parquet-compressed data format.

Experimental Setup. The experiments are conducted on a cluster of 11 nodes. Each node has a Westmere processor with a dual socket Intel(R) Xeon(R) X5660 CPU (6 cores per socket @ 2.80GHz), equipped with 64KB of L1 cache and 256KB L2 cache per core, 12MB of L3 cache shared, 48GB of RAM, and a RAID-0 of seven 250GB 7500 RPM SATA disks. The cluster runs Spark 2.1.0 and Hadoop HDFS 3.0.1. Spark launches 11 workers, each using 24 cores and 40GB of memory. We distribute all data across the 11 nodes with replication factor 3. All queries

[2] BlinkDB requires all queries to be known a priori, in order to decide on the samples. Therefore, we assumed the existence of an oracle that provides all queries to BlinkDB at initialization time. Clearly, this assumption strongly favors BlinkDB in the comparison.

[3] We used 18 out of the 22 TPC-H templates (Q_2 is not approximable, Q_4, Q_{21} and Q_{22} include `EXISTS` statement which require key of dimension relation thus no gain from approximation).

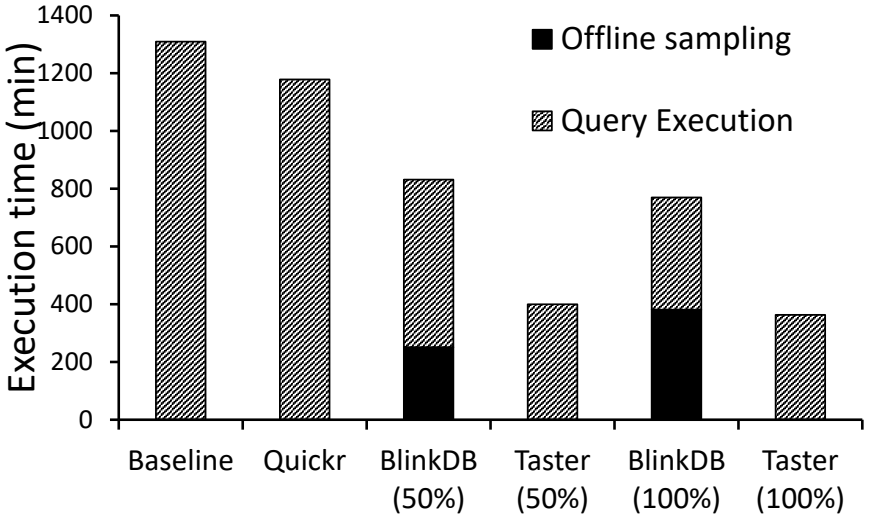

Figure 4.4 – TPC-H workload

are configured to return relative aggregation error per group less than 10%, and no missing groups. Finally, all queries are run from cold OS caches.

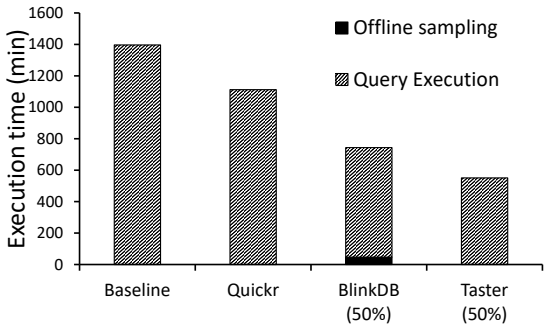

Figure 4.5 – TPC-DS workload

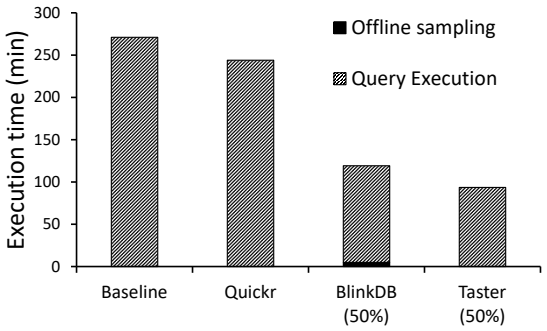

Figure 4.6 – instacart workload

Implementation. To have a fair comparison, we integrated all systems to SparkSQL 2.1.0, and extended the Catalyst built-in optimizer accordingly. For Quickr, we implemented the three sampler operators (Distinct, Uniform, Universe) and added all rules described in [70] to Catalyst. For BlinkDB, we followed the algorithms described in [6] to choose the same set of samples that the mixed integer linear program would select for the different workloads. We then generated the samples and executed the queries over that set of samples. Taster was implemented in Scala, over SparkSQL. We integrated Taster's tuner and optimization rules, as well as rudimentary costing capabilities into Spark Catalyst. Both query planner and tuner are centralized and run locally on the driver node of the Spark cluster. We implemented Taster's sketch-join algorithm using the serializable implementation of count-min sketch native to Spark 2.1.0. The uniform sampler is also native to Spark 2.1.0. The distinct sampler operator was implemented as an additional operator over DataFrames, using the algorithm described

in Section 2.3. The error estimator for samplers was estimated as described in Section 4.3.2. For robustness and scalability, all data, metadata, and materialized intermediate summaries of Taster were stored in HDFS, except of the in-memory buffer, which was implemented as persisted RDDs.

Statistics, Plan and Data storage. In order to allow scale-out computation under heavy load and be resilient to node failures, Taster serializes and stores all data, metadata, and materialized intermediate summaries in HDFS. The in-memory buffer is comprised by persisted RDDs.

4.5.1 Comparison to state-of-the-art AQP engines

We first evaluate the end-to-end performance of Taster compared to the state-of-the-art AQP systems.

Methodology. To compare all systems in a variety of workloads, we execute query sequences over all three datasets. To emulate workload shifts and examine system adaptivity, we instantiate 200 queries from the benchmark templates and issue them in random order. For each benchmark we randomly choose one of the available templates with equal probability (uniformly) and generate a new query by randomly choosing the predicate value. For TPC-H, both Taster and BlinkDB are tested with storage budgets 50% and 100% of the size of the compressed dataset. For TPC-DS and instacart, the queries have fewer prospective stratification attribute sets and require less space for samples. Therefore, we present results only for the 50% storage budget.

End-to-end execution time. Figures 4.4,4.5, and 4.6 present the required time for executing all 200 queries for each of the workloads. The reported time includes initialization time (i.e., the creation of the samples for BlinkDB). As expected, BlinkDB with only 50% budget requires less time for constructing the samples, but incurs a higher execution time since less queries are approximable by the set of available samples. Specifically, for TPC-H (Figure 4.4), BlinkDB 50% offers 2.25× speed-up compared to the Baseline, and requires 251 seconds for pre-computing the sample, whereas BlinkDB 100% offers 3.36× performance increase but spends 380 seconds on sampling. Quickr requires no preprocessing, but offers a smaller performance boost (1.2×). This is attributed mainly to the relevantly shallow queries of TPC-H, as well as the small network congestion of the cluster. Finally, Taster achieves low response time and ~ 3× speed-up without pre-computing the samples, by adapting to the query workload. We also see that Taster with 50% and 100% storage budget have a similar performance (difference is less than 10%), precisely because the system adapts to the workload and does not require all synopses to be present at all times.

The results with TPC-DS and instacart workloads (Figures 4.5,4.6) were qualitatively similar, confirming the applicability of Taster to different data and workload characteristics. In

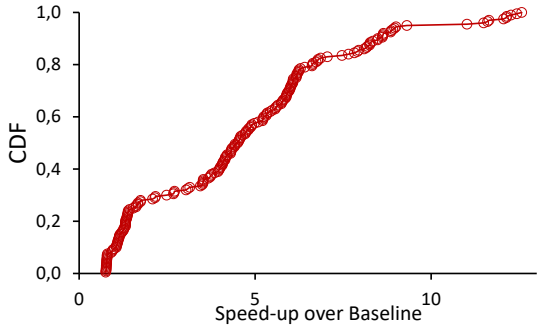

Figure 4.7 – Individual performance gains

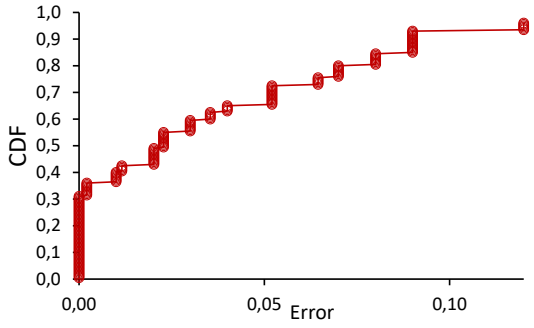

Figure 4.8 – Approximation error

particular, Taster has slightly better performance from BlinkDB, yet without requiring any initialization time. For TPC-DS, this performance improvement is attributed mainly to the capability of Taster to summarize also intermediate results (specifically, the join between tables $store\_sales$ and $date\_dim$, which appears frequently in the workload), rather than only base relations. For instacart, the increased performance of Taster comes from the extensive use of sketches.

Individual performance gains for TPC-H queries. Fig. 4.7 presents a CDF of the speed-up of Taster for TPC-H queries. Taster slows down less than 10% ($\sim 0.8\times$) of the queries, mostly due to the planning and tuning overhead, as well as the small overhead of online sampling. However, more than 50% of the queries are being sped-up more than $6\times$. The maximum speed-up ($13\times$) is achieved using sketches.

Approximation error for TPC-H queries. We also verified that the approximations of Taster are within the desired accuracy requirements, with high probability. Figure 4.8 presents a CDF of the observed aggregation error, for the TPC-H queries. The user requirements for these experiments are: (a) all groups should be detected, and (b) aggregate error should be less than 10%. By employing distinct sampling with stratification guarantees, Taster misses no groups. Furthermore, more than 93% of the queries have error less than 10%, and all queries have error less than 12%. These numbers are very close to the accuracy achieved from BlinkDB with offline sampling.

Summary. Taster substantially outperforms Quickr and offers comparable performance to BlinkDB, yet without requiring a priori knowledge of the workload, and without an offline sample pre-computation. Hence, Taster enables instant access to data while adhering to user accuracy requirements.

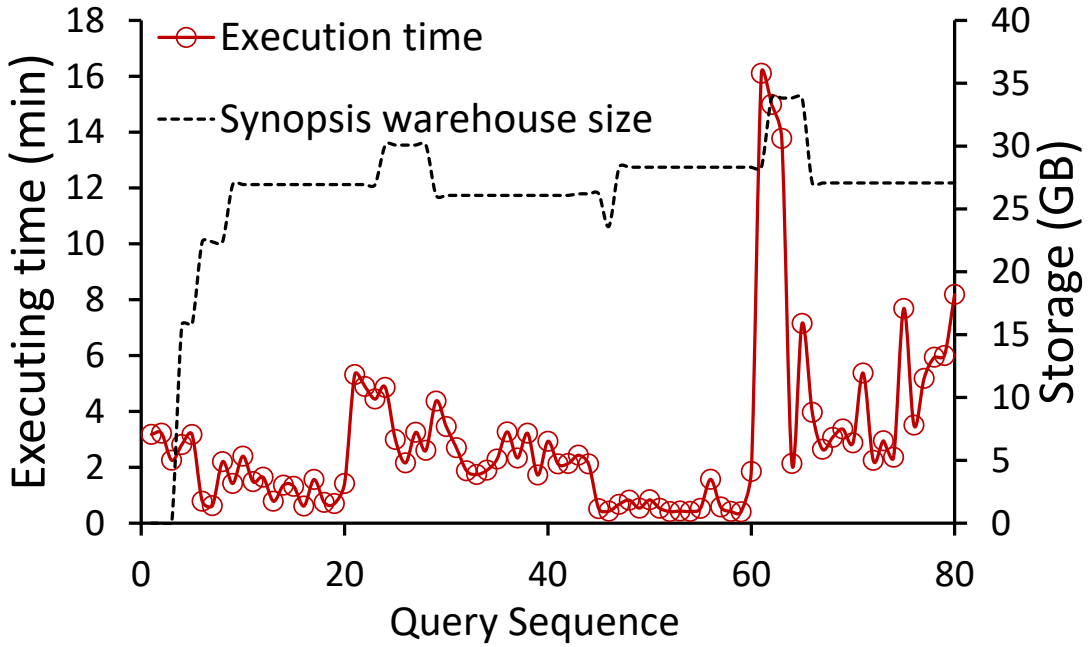

Figure 4.9 – Taster adapting to query workload

4.5.2 Adapting to query workload

Methodology. In this experiment, we evaluate the robustness of Taster to workload shifts, i.e., changes in query stratification attributes, the accessed tables, and query predicates. To emulate a real world scenario, we execute a sequence of 80 TPC-H queries, generated from the 18 used query templates by varying the filtering predicates. We split the queries into 4 *epochs* of 20 queries each, based solely on the query execution time, i.e., queries in each group have similar execution time when executed using Baseline. The following templates are used per epoch: (1): q6, q14, q17 (2): q5, q8, q11, q12 (3): q1, q3, q16, q19 (4): q7, q9, q13, q18. As the grouping relies only on query execution time, the queries within each epoch may use different synopses. For example, in epoch (2) template of q5 requires a synopsis with stratification on $orderkey$ whereas template of q8 requires stratification on $partkey$. The storage budget for Taster is set to 35GB.

Figure 4.9 presents the execution time and storage requirements of Taster at each query. Taster's tuner continuously re-evaluates the synopses stored in the synopsis warehouse, and it frequently drops and build some synopses while executing the queries. At the beginning of each epoch, Taster quickly recognizes the new useful synopses, and makes space for them by evicting the older ones. During the last epoch, the tuner decides to materialize the synopses earlier, since the new synopses provide a higher prospective gain. During the transition from the second to the third epoch, as the queries of the third epoch have similar performance with the approximated queries of the second epoch, in the plot it seems as if all the queries are approximated which is not valid.

Summary. Taster adapts the available synopses to the evolving workload. This enables better space utilization with performance comparable to state-of-the-art offline AQP systems.

4.5.3 Adapting the sliding window length to query workload

Methodology. We now evaluate the adaptivity of the tuner in terms of the sliding window length w used for predicting the future queries. We execute a sequence of 200 TPC-H queries, generated by using the 18 query templates. The queries are executed in random order. To evaluate the impact of the adaptive sliding window, the same query workload is executed using three static configurations ($w = 5$, $w = 10$, and $w = 50$), and the adaptive configuration where w changes according to the queries. Storage budget is fixed to 35GB.

Figure 4.10 presents the cumulative execution time for all queries, for the considered configurations. Taster with adaptive sliding window length starts with window size 5 and increases/decreases according to the correctness of prior predictions. During this experiment the window size fluctuates between 12 and 17, but never converges. This exemplifies the need for an adaptive sliding window length. Among the static window configurations, Taster with window size 10 performs the best, but it is still noticeably slower than the adaptive version. Window sizes 5 and 50 lead to fairly bad performance, i.e., the predictive power of the tuner for future queries is annihilated.

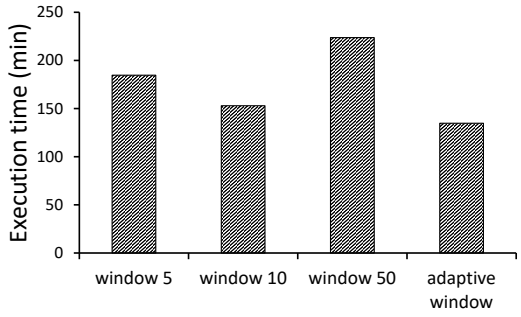

Figure 4.10 – Varying the horizon size

Figure 4.11 – Varying the storage budget

4.5.4 Storage elasticity

Methodology. We now investigate how Taster adapts to changing storage budget. We run a sequence of 250 TPC-H queries in random order, progressively changing the storage budget configuration. The queries are executed as a sequence without pause – even at the times that the storage budget is changed. The budget allocated for each set of queries is (relative to the size of the compressed data size): first set 20%, second set 50%, third set 100%, firth set 50% and finally fith set 100%.

Adapting to Storage. Figure 4.11 presents the average speed-up for these storage configura-

tions compared to Baseline. With 20% of storage, Taster fits only one sample and a sketch, thereby providing very limited approximation potentials. When given 50%, Taster has sufficient space to keep *almost* all synopses, whereas a budget of 100% enables Taster to keep all synopses. When storage allowance is reduced, Taster automatically invokes the tuner to keep the synopses that will maximize the gain, thereby minimizing the performance impact. Assuming that BlinkDB would be able to adapt to decreasing storage budget by dropping samples, it would be unable to build them when more storage becomes available. It would require blocking execution until constructing the required samples to speed-up queries.

Summary. Taster adapts to dynamic storage budget through dropping and re-computing synopses as byproducts of query execution. This approach enables Taster to speed-up queries even under stringent storage policies, as well as instantly speed-up queries when the user increases storage resources.

4.5.5 Utilizing user hints

The final experiment focuses on examining how Taster utilizes user hints to improve performance. The experiment simulates the following scenario: the user already has an idea on the analysis that will be conducted *on one part* of the database (on a subset of the tables) and she advices Taster on the samples that need to be taken, e.g., by listing representative queries, or even by explicitly stating the required samples. In this case, Taster constructs and pins the synopses in the synopsis warehouse offline, and manages the remaining quota online for storing new synopses. We demonstrate this setup by generating two databases – two instances of TPC-H (scale factor 300) – and using Taster to query both, with intervening queries. For the first database, db_{off}, we instruct Taster at initialization for the synopses that need to be created offline (in this case, samples on the $lineitem$ table). For the second, db_{onl}, we let Taster generate and handle the synopses online. Taster is also free to create additional samples for db_{off}, if the precomputed samples do not cover all queries.

Offline sampling in db_{off} follows the state-of-the-art variational subsampling approach of VerdictDB [96]. Notice that this approach requires the following offline steps: (a) creating a shuffled clone of the $lineitem$ table (the scrambled copy), and (b) extracting the samples. We also alter the query execution process to apply variational subsampling. Both databases are queried with 100 queries of TPC-H, i.e., a total of 200 queries, in mixed

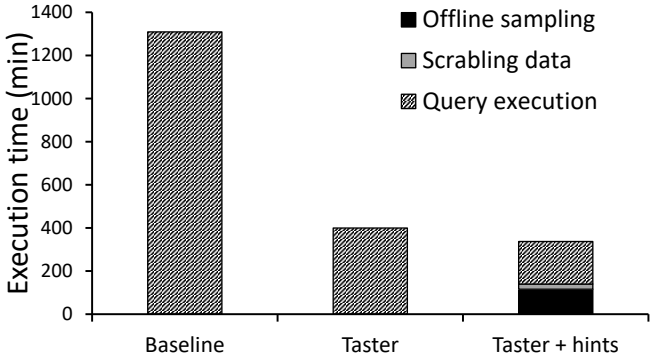

Figure 4.12 – Performance with user hints

sketch-1	*order_id*, *count*(∗) FROM *orderproducts* JOIN *orders* WHERE *o_order_dow* = *_day_* AND *o_order_hod* > *_hour_*
sketch-2	*product_id*, *count*(∗) FROM *orderproducts* JOIN *products* WHERE *p_product_name* = *_productname_*
sketch-3	*product_id*, *count*(∗) FROM *orderproducts* JOIN *products* JOIN *departments* WHERE *d_department* = *_department_*
sketch-4	*product_id*, *count*(∗) FROM *orderproducts* JOIN *products* JOIN *aisles* WHERE *a_aisle* = *_aislename_*
sample-1	*product_id*, *count*(∗) FROM *orderproducts* JOIN *orders* WHERE *o_order_dow* = *_day_* AND *o_order_hod* > *_hour_*
sample-2	*order_id*, *count*(∗) FROM *orderproducts* JOIN *products* WHERE *p_product_name* = *_productname_*
sample-3	*order_id*, *count*(∗) FROM *orderproducts* JOIN *products* JOIN *departments* WHERE *d_department* = *_department_*
sample-4	*order_id*, *count*(∗) FROM *orderproducts* JOIN *products* JOIN *aisles* WHERE *a_aisle* = *_aislename_*

Table 4.1 – Instacart micro-benchmark queries. Variables starting and ending with _ are randomly set for query variation.

order. For this experiment, Taster is
given a total of 50 GB for synopsis
quota (since the scrambled table is only used offline, we do not include it in the quota).

Figure 4.12 presents the time spent for answering all 200 queries (denoted as Taster + hints), as well as the time spent in the offline phase. For comparison, the figure also includes the elapsed time for getting exact results (Baseline), and the time for executing the same workload in Taster without hints (Taster). Clearly, hints help Taster to increase query performance, by taking the sampling phase offline. Furthermore, the use of variational subsampling enables the use of smaller samples. In particular, the average speed-up over all queries was 12.6× compared to the baseline, and 4.98× compared to Taster without hints. The speed-up over the queries on db_{off} only (these are the queries using the pre-computed samples) was 20.43× and 9.24× compared to Taster without hints. The construction of the samples using variational subsampling, however, takes a non-negligible amount of pre-processing (116 minutes), delaying the first insights from the dataset. Therefore, a hints-driven offline phase is beneficial when the user knows that a database/table will be frequently queried in the near future; it reduces both query execution time and the size of the generated samples.

4.6 Conclusion

Approximate query processing engines – both offline and online – gained significant interest in the last years, as they offer low latency data analytics in return to an acceptable, slightly relaxed precision of the results. However, the ever-growing data sizes combined with the need

of today's data scientists to get immediate insights out of big data, introduce a new set of challenges to these systems. On the one hand, offline approximation approaches require long pre-processing and knowledge of the expected workload, and have high storage requirements. On the other hand, online approaches require reading all data for each query in order to collect samples, hence offering much smaller performance gains. In this chapter, we demonstrate Taster, a system that adaptively combines the two approaches. Synopses in Taster summarize both base relations and intermediary results (frequent subplans). They are generated in an online fashion, as byproducts of the queries, but they can also be saved and reused across several queries similar to offline AQP engines. Importantly, the stored synopses transparently adapt to the ever-shifting user workload, without user intervention, and without requiring a priori knowledge of the query workload. Finally, Taster can also integrate hints, e.g., for creating some samples offline, thereby further reducing query latency. A thorough evaluation of Taster using three industry-standard benchmarks demonstrates that it adapts to variations in workload and storage, and it outperforms both online and offline AQP approaches, without requiring a priori knowledge of the query workload.

5 The Big Picture

The desired interactive nature and effectiveness of data exploration tasks are hindered by a number of obstacles. Having to go through effectively never-before-seen data, or to identify new patterns in a dataset, implies that loading entire datasets in a DBMS and investing on auxiliary structures a priori may very well be an investment that does not pay off. Ideally, a data scientist would launch her queries over raw data, turning the data-to-query time to zero. Then, to further optimize for her use case, once she places her focus in a specific area of the dataset, a system should automatically construct indexing structures to speed up queries over this area.

The research of this thesis stems from the need of modern applications to explore data quickly and efficiently. We identify that the assumption of conventional approaches for a priori access to data and predictable workloads is inapplicable in modern data exploration applications. Furthermore, the ever-increasing datasets require more low-latency storage which becomes more costly to maintain. To address these findings, we re-designed a database architecture to enable data exploration, focusing on tuning data structures to minimize data access while removing the need pre-processing and taking maximum advantage of available storage and compute resources.

This thesis is part of a bigger agenda towards adaptive and self-tuning database systems aiming to reduce the cumulative cost of data access for data exploration. The ultimate goal is enabling the user to explore never-before-seen datasets, stored in a variety of data formats and heterogenous data sources, at real-time requiring no preprocessing while being able to take full advantage of available resources. This chapter summarizes the contributions of this thesis and discusses a number of ongoing efforts to address open challenges to adaptive data access methods.

5.1 Adaptive Data Access: What we did

Each chapter of this thesis moves toward the direction of reducing the cost of query execution for exploration applications. To achieve that, we aim to reduce data access while assuming no a priori knowledge of workload or dataset and requiring no pre-processing time. The lack of knowledge about the workload and dataset limits optimization decisions and combined with the lack of pre-processing constraints physical tuning.

To remove pre-processing and enable instant access to data, we utilize the in-situ query processing paradigm to execute queries over raw data files. To improve optimization decisions as well as to make tuning decisions we collect statistics as by-product of query execution as the workload unfolds. Based on the collected information we tune access paths adaptively using two approaches depending on the accuracy requirements of the user. Specifically, when the user requires precise results we propose adapting to data distribution. This approach takes advantage of *implicit clustering* within data to improve filtering. Recognizing possible clustering in data, enables pre-filtering and reduces the size of indexes [14, 89, 90]. On the other hand, when the user accepts approximate results, we propose to speed-up queries through online approximation enhanced with intermediate result recycling. Specifically, we propose making data summaries a first-class citizen in query optimization and enable the materialization of summaries of intermediate query results which could be re-used by future queries [91].

This thesis pushes towards modern data exploration by studying the impact of workload and data-aware query execution. We discuss the importance of adaptivity in exploratory workloads on both query performance and storage budget and present techniques extending data management to embrace shifting workloads and unknown dataset and enable interactive data exploration.

5.2 Adaptive Data Access: Next Steps

Sortedness-aware index design. Modern systems and applications need to face new challenges, as ever-increasing volumes are increasing daily. Database management systems build indexes to reduce data accesses and speed-up look-ups. However, the constant influx of data requires novel indexes which are optimized for updates. As discussed in Chapter 3, data tends to have some explicit and implicit order, even without having to sort it. Such ordering schemes could be exploited to increase the insertion speed, as this information can be used to locate the insertion position faster. Different indexing solutions have tried to reduce the update over- head when inserting elements. Basic log data structures can achieve an insertion in $O(1)$. However, such data structures are not competitive at all when one wants to read data, with a worst-case complexity of $O(N)$. The LSM-tree is a data structure based on the log data structure, with an $O(1)$ amortized cost when inserting an element to it. It also includes some tree components to allow a faster search, such as levels, and multiple runs per level. On the

other hand, B^+ tree have a small read overhead, with a $O(\log(N))$ cost. Such data structures are then very good for read intensive scenarios, however they lag in write-intensive scenarios. B^+ tree insertions are performed in $O(\log(N))$, a big cost compared to LSM-trees insertion costs.

We propose optimizations over B^+ tree to take advantage of sorting properties of the underlying data, and provide update performance on par with LSM-trees. Specifically, we propose combining B^+ tree bulkloading by batching insertions into the B^+ tree and keeping pointers to most recently updated tree leaf nodes. Finally, by delaying the propagation of branch splits by introducing spill pages, we can further improve B^+ tree insertions performance while keeping look-up constant.

Extending utilization of approximate operators in Query Engines. The work discussed in Chapter 4 offers the platform to expand approximate query processing in multiple ways. First, expanding the arsenal of approximate operators enables Taster to further speed-up queries while increases the number of options thus increasing the cost of decision. Prospective operators are the HyperLogLog operator which enables the calculation of distinct values in a set as well as the integration of the Bloom filter to implement a Bloom-Join. Second, the window based prediction strategy designed in Chapter 4 exhibits positive results however the industrial workloads have limited variability in synopses options. In a real life workload such an approach may present different results. Thus, we want to study the performance of our algorithm on a real-world dataset and adjust the prediction algorithm either using a different variant of window forecasting or building a Machine Learning model.

Adaptive RDMA and NUMA aware task and data placement. Due to the ever-increasing datasets and the need for interactive analytics, modern database management systems require increasingly more memory. However, physical limits constrain memory size utilized by a server and reduce the scalability of data analysis systems [15].

The introduction of RDMA (Remote Direct Memory Access) enables a server to access memory of a remote server while avoiding the overheads of the network stack. This creates opportunities to reduce the cost of data access for distributed database management systems. A number of novel research techniques take advantage of the new hardware capabilities and propose updates to existing database operators and distributed storage designs.

Furthermore, modern processor vendors in order to achieve scalability, connect multiple sockets of multi-core processors. In this design, memory is decentralized, each socket having his own memory, forming a non-uniform memory access (NUMA) architecture. Specifically, accessing memory connected to a remote socket has higher latency and smaller bandwidth than accessing memory local to the socket. In addition, in this design, the network adapter is directly connected to one of the sockets. As a consequence the NUMA architecture has a direct impact on network accesses leading to Non-Uniform I/O Access (NUIOA). NUIOA-remote RDMA accesses must cross both the network interconnect, and the server interconnect.

This entails two levels of networks: an *intra-server network* and an *inter-server network* leading to multiple access paths with different bottlenecks. We propose to analyze the performance of different task and data placement strategies for query execution and subsequently design an adaptive algorithm monitoring the load on processing resources, network connections, memory buses and QuickPath interconnects. Based on the collected load statistics, the adaptive algorithm will decide where (i) to store data, and (ii) where to execute operations. Initial results have shown that by reducing the number of buses a data connection has to cross, is key to the performance of an operator. However, if either compute resources or memory bus of a server is fully utilized, additional request will slow-down overall execution. Thus, by partitioning and replicating data, the system may reduce the contention of specific resources and further improve system throughput.

Adaptive query processing using data source oriented scheduling. Applications of exploratory nature despite employing adaptive indexing or approximation techniques to reduce data access still aggregate vast amounts of data which have to be stored and accessed efficiently. The data is stored in a variety of formats ranging from raw data files to relational databases and on a variety of storage devices. Each data source, has different performance characteristics and access latency. For example, data stored on SSD/DRAM low-latency devices is accessed faster than data stored on SATA HDD high-density capacity devices or tape devices.

Furthermore, meaningful data analysis rely on analyzing and combining information from an increasing number of datasets which due to the growth in data volume are stored in multiple data sources. However, due to the access latency mismatch between data sources, the performance of analysis tasks which access different data sources will be always bounded by the latency of the slowest data source.

We propose a novel approach for scale-out query execution over systems accessing data sources with heterogeneous characteristics. We propose a scheduler which prioritizes the execution of queries whose data is already available in low latency data sources and asynchronously initiates data transfer calls for queries accessing data stored on high latency sources. The approach aims at maximizing resource utilization and query throughput irrespective of storage hardware characteristics thus offering a cost-effective solution for data analytics.

Bibliography

[1] The Instacart Online Grocery Shopping Dataset 2017. https://www.instacart.com/datasets/grocery-shopping-2017. accessed 28-May-2018.

[2] C. L. Abad, N. Roberts, Y. Lu, and R. H. Campbell. A storage-centric analysis of MapReduce workloads: File popularity, temporal locality and arrival patterns. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 100–109, 2012.

[3] A. Abouzied, D. J. Abadi, and A. Silberschatz. Invisible loading: access-driven data transfer from raw files into database systems. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 1–10, 2013.

[4] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional Samples for Approximate Answering of Group-by Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 487–498, 2000.

[5] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The Aqua Approximate Query Answering System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 574–576, 1999.

[6] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 29–42, 2013.

[7] S. Agrawal, E. Chu, and V. Narasayya. Automatic Physical Design Tuning: Workload As a Sequence. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 683–694, 2006.

[8] S. Agrawal, V. R. Narasayya, and B. Yang. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 359–370, 2004.

[9] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 169–180, 2001.

[10] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 241–252, 2012.

[11] A. A. Alamoudi, R. Grover, M. J. Carey, and V. R. Borkar. External Data Access And Indexing In AsterixDB. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, pages 3–12, 2015.

[12] K. Alexiou, D. Kossmann, and P.-Å. Larson. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. *Proceedings of the VLDB Endowment*, 6(14):1714–1725, 2013.

[13] N. Alon, Y. Matias, and M. Szegedy. The Space Complexity of Approximating the Frequency Moments. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC)*, pages 20–29, 1996.

[14] A. Anagnostou, M. Olma, and A. Ailamaki. Alpine: Efficient In-Situ Data Exploration in the Presence of Updates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1651–1654, 2017.

[15] R. Appuswamy, M. Olma, and A. Ailamaki. Scaling the Memory Power Wall With DRAM-Aware Data Management. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, pages 3:1–3:9, 2015.

[16] I. Arapakis, X. Bai, and B. B. Cambazoglu. Impact of response latency on user behavior in web search. In *Proceedings of the Annual ACM SIGIR International Conference on Research and Development in Information Retrieval*, pages 103–112, 2014.

[17] M. Athanassoulis and A. Ailamaki. BF-Tree: Approximate Tree Indexing. *Proceedings of the VLDB Endowment*, 7(14):1881–1892, 2014.

[18] M. Athanassoulis and S. Idreos. Design Tradeoffs of Data Access Methods. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Tutorial*, 2016.

[19] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing Access Methods: The RUM Conjecture. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 461–466, 2016.

[20] M. Athanassoulis, Z. Yan, and S. Idreos. UpBit: Scalable In-Memory Updatable Bitmap Indexing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2016.

[21] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 539–550, 2003.

[22] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani. Parallel data analysis directly on scientific file formats. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 385–396, 2014.

[23] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[24] R. Borovica-Gajic, S. Idreos, A. Ailamaki, M. Zukowski, and C. Fraser. Smooth Scan: Statistics-oblivious access paths. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 315–326, 2015.

[25] P. B. Borwein. On the Complexity of Calculating Factorials. *Journal of Algorithms*, 6(3):376–380, 1985.

[26] N. Bruno and S. Chaudhuri. An Online Approach to Physical Design Tuning. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 826–835, 2007.

[27] U. Çetintemel, M. Cherniack, J. DeBrabant, Y. Diao, K. Dimitriadou, A. Kalinin, O. Papaemmanouil, and S. B. Zdonik. Query Steering for Interactive Data Exploration. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2013.

[28] G. Chatzopoulou, M. Eirinaki, and N. Polyzotis. Query Recommendations for Interactive Database Exploration. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 3–18, 2009.

[29] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. R. Narasayya. Overcoming Limitations of Sampling for Aggregation Queries. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 534–542, 2001.

[30] S. Chaudhuri, G. Das, and V. Narasayya. A Robust, Optimization-based Approach for Approximate Answering of Aggregate Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 295–306, 2001.

[31] S. Chaudhuri and V. R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 146–155, 1997.

[32] S. Chaudhuri and V. R. Narasayya. AutoAdmin 'What-if' Index Analysis Utility. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 367–378, 1998.

[33] Y. Chen, S. Alspaugh, and R. H. Katz. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, 2012.

[34] Y. Cheng and F. Rusu. Parallel in-situ data processing with speculative loading. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1287–1298, 2014.

[35] J. C.-Y. Chou, M. Howison, B. Austin, K. Wu, J. Qiang, E. W. Bethel, A. Shoshani, O. Rübel, Prabhat, and R. D. Ryne. Parallel index and query for large scale data analysis. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 30:1—-30:11, 2011.

[36] C. J. Clopper and E. S. Pearson. The Use of Confidence or Fiducial Limits Illustrated in the Case of the Binomial. *Biometrika*, 26(4):404–413, 1934.

[37] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends in Databases*, 4(1-3):1–294, 2012.

[38] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[39] J. R. Dabrowski and E. V. Munson. Is 100 Milliseconds Too Fast? In *Proceedings of the ACM SIGCHI Extended Abstracts on Human Factors in Computing Systems*, pages 317–318, 2001.

[40] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic SQL tuning in oracle 10g. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1098–1109, 2004.

[41] D. J. DeWitt, A. Halverson, R. V. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split query processing in polybase. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1255–1266, 2013.

[42] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, and G. Wood. Automatic Performance Diagnosis and Tuning in Oracle. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 84–94, 2005.

[43] B. Ding, S. Huang, S. Chaudhuri, K. Chakrabarti, and C. Wang. Sample + Seek: Approximating Aggregates with Distribution Precision Guarantee. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 679–694, 2016.

[44] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[45] S. J. Finkelstein, M. Schkolnick, and P. Tiberio. Physical Database Design for Relational Databases. *ACM Transactions on Database Systems (TODS)*, 13(1):91–128, 1988.

[46] P. Flajolet and G. N. Martin. Probabilistic Counting Algorithms for Data Base Applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[47] C. Furtado, A. A. B. Lima, E. Pacitti, P. Valduriez, and M. Mattoso. Physical and Virtual Partitioning in OLAP Database Clusters. In *Proceedings of the Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 143–150, 2005.

[48] A. Galakatos, A. Crotty, E. Zgraggen, C. Binnig, and T. Kraska. Revisiting Reuse for Approximate Query Processing. *Proceedings of the VLDB Endowment*, 10(10):1142–1153, 2017.

[49] V. R. Gankidi, N. Teletia, J. M. Patel, A. Halverson, and D. J. DeWitt. Indexing HDFS Data in PDW: Splitting the data from the index. *Proceedings of the VLDB Endowment*, 7(13):1520–1528, 2014.

[50] J. Goldstein and P.-A. Larson. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 331–342, 2001.

[51] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 371–381, 2010.

[52] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 209–218, 1993.

[53] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. J. DeWitt, and G. Heber. Scientific data management in the coming decade. *ACM SIGMOD Record*, 34(4):34–41, 2005.

[54] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE: A Main Memory Hybrid Storage Engine. *Proceedings of the VLDB Endowment*, 4(2):105–116, 2010.

[55] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *Proceedings of the VLDB Endowment*, 5(6):502–513, 2012.

[56] T. Härder. Selecting an Optimal Set of Secondary Indices. In *Proceedings of the European Cooperation in Informatics (ECI)*, pages 146–160, 1976.

[57] J. He. Advances in Data Mining: History and Future. In *Proceedings of the International Symposium on Intelligent Information Technology Application*, pages 634–636, 2009.

[58] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 171–182, 1997.

[59] G. Hu, J. Ma, and B. Huang. High throughput implementation of MD5 algorithm on GPU. In *Proceedings of the Inte Conference on Ubiquitous Information Technologies & Applications (ICUT)*, pages 1–5, 2009.

[60] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 57–68, 2011.

[61] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2007.

[62] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing Tuple Reconstruction in Column-Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 297–308, 2009.

[63] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *Proceedings of the VLDB Endowment*, 4(9):586–597, 2011.

[64] S. Idreos, O. Papaemmanouil, and S. Chaudhuri. Overview of Data Exploration Techniques. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Tutorial*, pages 277–281, 2015.

[65] M. Ivanova, M. L. Kersten, and S. Manegold. Data Vaults: A Symbiosis between Database Technology and Scientific File Repositories. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 485–494, 2012.

[66] I. iView. The digital universe in 2020: Big data. bigger digital shadows, and biggest growth in the far east. availableathttps://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf.

[67] C. M. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the DBO engine. *ACM Transactions on Database Systems (TODS)*, 33(4), 2008.

[68] A. Jindal and J. Dittrich. Relax and Let the Database Do the Partitioning Online. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 65–80, 2011.

[69] S. Kandula. Errata and Proofs for Quickr. Technical report, 2016.

[70] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 631–646, 2016.

[71] Y. Kargin, M. L. Kersten, S. Manegold, and H. Pirk. The DBMS - your big data sommelier. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 1119–1130, 2015.

[72] A. R. Karlin, M. S. Manasse, L. A. McGeoch, and S. S. Owicki. Competitive Randomized Algorithms for Non-Uniform Problems. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 301–309, 1990.

[73] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast Queries Over Heterogeneous Data Through Engine Customization. *Proceedings of the VLDB Endowment*, 9(12):972–983, 2016.

[74] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki. Just-In-Time Data Virtualization: Lightweight Data Management with ViDa. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015.

[75] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive Query Processing on RAW Data. *Proceedings of the VLDB Endowment*, 7(12):1119–1130, 2014.

[76] M. Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, 2010.

[77] M. L. Kersten, S. Idreos, S. Manegold, and E. Liarou. The Researcher's Guide to the Data Deluge: Querying a Scientific Database in Just a Few Seconds. *Proceedings of the VLDB Endowment*, 4(12):1474–1477, 2011.

[78] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015.

[79] B. Kovalerchuk and J. Schwing. *Visual and Spatial Analysis: Advances in Visual Data Mining, Reasoning and Problem Solving*. 01 2005.

[80] N. Laptev, K. Zeng, and C. Zaniolo. Early Accurate Results for Advanced Analytics on MapReduce. *Proceedings of the VLDB Endowment*, 5(10):1028–1039, 2012.

[81] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, C. Faloutsos, J. VanBriesen, and N. Glance. Cost-effective Outbreak Detection in Networks. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 420–429, 2007.

[82] S. Lightstone, T. J. Teorey, and T. P. Nadeau. *Physical Database Design: the database professional's guide to exploiting indexes, views, storage, and more*. Morgan Kaufmann, 2007.

[83] Z. Liu and J. Heer. The Effects of Interactive Latency on Exploratory Visual Analysis. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2122–2131, 2014.

[84] S. Lohr. *Sampling: Design and Analysis*. 2009.

[85] F. López-Blázquez and B. S. Mino. Binomial Approximation to Hypergeometric Probabilities. *Journal of Statistical Planning and Inference*, 87(1):21–29, 2000.

[86] G. Marchionini. Exploratory Search: From Finding to Understanding. *Communications of the ACM*, 49(4):41–46, 2006.

[87] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. *Proceedings of the VLDB Endowment*, 3(1):330–339, 2010.

[88] G. Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 476–487, 1998.

[89] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Adaptive Partitioning and Indexing for in-situ Query Processing. *The VLDB Journal*, 2017.

[90] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Slalom: Coasting Through Raw Data via Adaptive Partitioning and Indexing. *Proceedings of the VLDB Endowment*, 10(10):1106–1117, 2017.

[91] M. Olma, O. Papapetrou, R. Appuswamy, and A. Ailamaki. Taster: Self-Tuning, Elastic and Online Approximate Query Processing. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2019.

[92] P. E. O'Neil. Model 204 Architecture and Performance. In *Proceedings of the International Workshop on High Performance Transaction Systems (HPTS)*, pages 40–59, 1987.

[93] N. Pansare, V. Borkar, C. Jermaine, and T. Condie. Online Aggregation for Large MapReduce Jobs. *Proceedings of the VLDB Endowment*, 4(4):1135–1145, 2011.

[94] S. Papadomanolakis and A. Ailamaki. AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, page 383, 2004.

[95] O. Papapetrou, W. Siberski, and W. Nejdl. Cardinality Estimation and Dynamic Length Adaptation for Bloom Filters. *Distrib. Parallel Databases*, 28(2-3):119–156, 2010.

[96] Y. Park, B. Mozafari, J. Sorenson, and J. Wang. VerdictDB: Universalizing Approximate Query Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1461–1476, 2018.

[97] K. Pearson. Contributions to the Mathematical Theory of Evolution. II. Skew Variation in Homogeneous Material. *Philosophical Transactions of the Royal Society of London*, 186(Part I):343–424, 1895.

[98] J. Peng, D. Zhang, J. Wang, and J. Pei. AQP++: Connecting Approximate Query Processing With Aggregate Precomputation for Interactive Analytics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1477–1492, 2018.

[99] W. W. Peterson and D. T. Brown. Cyclic Codes for Error Detection. *Proceedings of the IRE*, 49(1):228–235, 1961.

[100] E. Petraki, S. Idreos, and S. Manegold. Holistic Indexing in Main-memory Column-stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2015.

[101] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 3rd edition, 2002.

[102] S. Richter, J.-A. Quiané-Ruiz, S. Schuh, and J. Dittrich. Towards zero-overhead static and adaptive indexing in Hadoop. *The VLDB Journal*, 23(3):469–494, 2013.

[103] R. L. Rivest. The MD5 Message-Digest Algorithm. *RFC*, 1321:1–21, 1992.

[104] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. COLT: Continuous On-Line Database Tuning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 793–795, 2006.

[105] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The Uncracked Pieces in Database Cracking. *Proceedings of the VLDB Endowment*, 7(2):97–108, 2013.

[106] L. Sidirourgos and M. L. Kersten. Column Imprints: A Secondary Index Structure. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 893–904, 2013.

[107] L. Sidirourgos, M. L. Kersten, and P. A. Boncz. SciBORQ: Scientific data management with Bounds On Runtime and Quality. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 296–301, 2011.

[108] R. R. Sinha, S. Mitra, and M. Winslett. Bitmap indexes for large scientific data sets: a case study. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2006.

[109] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. Fine-grained Partitioning for Aggressive Data Skipping. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1115–1126, 2014.

[110] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[111] G. Valentin, M. Zuliani, D. C. Zilio, G. M. Lohman, and A. Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend its own Indexes. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 101–110, 2000.

[112] X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 19–35, 2005.

[113] E. Wu and S. Madden. Partitioning Techniques for Fine-grained Indexing. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 1127–1138, 2011.

[114] K. Wu, S. Ahern, E. W. Bethel, J. Chen, H. Childs, E. Cormier-Michel, C. Geddes, J. Gu, H. Hagen, B. Hamann, W. Koegler, J. Lauret, J. Meredith, P. Messmer, E. J. Otoo, V. Perevoztchikov, A. Poskanzer, O. Rübel, A. Shoshani, A. Sim, K. Stockinger, G. Weber, and W.-M. Zhang. FastBit: interactively searching massive data. *Journal of Physics: Conference Series*, 180(1):012053, 2009.

[115] K. Zeng, S. Agarwal, A. Dave, M. Armbrust, and I. Stoica. G-OLA: Generalized On-Line Aggregation for Interactive Analysis on Big Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 913–918, 2015.

[116] K. Zeng, S. Agarwal, and I. Stoica. iOLAP: Managing Uncertainty for Efficient Incremental OLAP. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1347–1361, 2016.

[117] K. Zeng, S. Gao, B. Mozafari, and C. Zaniolo. The Analytical Bootstrap: A New Method for Fast Error Estimation in Approximate Query Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 277–288, 2014.

[118] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1087–1097, 2004.

**Ce document a été imprimé au Centre d'impression EPFL,
imprimerie climatiquement neutre, certifiée myClimate.**