Big Data Exploration

By Stratos Idreos
CWI, Amsterdam, The Netherlands

Introduction

The Big Data Era. We are now entering the era of data deluge, where the amount of data outgrows the capabilities of query processing technology. Many emerging applications, from social networks to scientific experiments, are representative examples of this deluge, where the rate at which data is produced exceeds any past experience. For example, scientific analysis such as astronomy is soon expected to collect multiple Terabytes of data on a daily basis, while already web-based businesses such as social networks or web log analysis are confronted with a growing stream of large data inputs. Therefore, there is a clear need for efficient big data query processing to enable the evolution of businesses and sciences to the new era of data deluge.

In this chapter, we focus on a new direction of query processing for big data where data exploration becomes a first class citizen. Data exploration is necessary when new big chunks of data arrive rapidly and we want to react quickly, i.e., with little time to spare for tuning and set-up. In particular, our discussion focuses on database systems technology, which for several decades has been the predominant data processing tool.

In this chapter, we introduce the concept of data exploration and we discuss a series of early techniques from the database community towards the direction of building database systems which are tailored for big data exploration, i.e., adaptive indexing, adaptive loading and sampling-based query processing. These directions focus on reconsidering fundamental assumptions and on designing next generation database architectures for the big data era.

In Need for Big Data Query Processing

Let us first discuss the need for efficient query processing techniques over big data. We briefly discuss the impact of big data both in businesses and in sciences.

Big Data in Businesses. For businesses, fast big data analysis translates to better customer satisfaction, better services and in turn it may be the catalyst in creating and maintaining a successful business. Examples of businesses in need for analyzing big data include any kind of web and data-based IT business, ranging from social networks to e-commerce, news, emerging mobile data businesses, etc. The most typical example in this case is the need to quickly understand user behavior and data trends; this is necessary in order to dynamically adapt services to the user needs.

Businesses continuously monitor and collect data about the way users interact with their systems, e.g., in an e-commerce web site, in a social network, or in a GPS navigation system, and this data needs to be analyzed quickly in order to discover interesting trends. Speed here is of essence as these businesses get multiple Terabytes of data on a daily basis and the kinds of trends observed might change from day to day or from hour to hour. For example, social networks and mobile data applications observe rapid changes on user interests, e.g., every single minute there are 700,000 status updates on Facebook and 700.000 queries on Google. This results to staggering amounts of data that businesses need to analyze as soon as possible and while it is still relevant.

Big Data in Sciences. For sciences, fast big data analysis can push scientific discovery forward. All sciences nowadays struggle with data management, e.g., astronomy, biology, etc. At the same time, the expectation is that in the near future sciences will increase even more their ability to collect data. For example, the Large Synoptic Survey Telescope project in USA expects a daily collection of 20 Terabytes, while the Large Hadron Collider in CERN in Europe already creates an even bigger amount of data. With multiple Terabytes of data on a daily basis, data exploration becomes essential in order to allow scientists to quickly focus on data parts where there is a good probability to find interesting observations.

Big Data Challenges for Query Processing

We continue the discussion by focusing on the challenges that big data bring for state-of-the-art data management systems.

Existing Technology. Data management technology has a tremendous and important history of achievements and numerous tools and algorithms to deal with scalable data processing. Notable recent examples include column-store database systems (Stonebraker, et al. 2005) (Boncz, Zukowski and Nes 2005) and map-reduce systems (Dean and Ghemawat 2004) as well as recent hybrids that take advantage of both the structured database technology and the massively scalable map-reduce technology (Hadapt 2012) (Platfora 2012) (Abouzeid, et al. 2009). All small and major organizations rely on data management technology to store and analyze their data. Sciences, on the other hand, rely on a mix of data management technologies and proprietary tools that accommodate the specialized query processing needs in a scientific environment.

Big Data Challenges. Regardless of the kind of technology used, the fundamental problem nowadays is that we cannot consume and make sense of all these data fast enough. This is a direct side effect of some of the assumptions that are in inherit in modern data management systems.

First, state-of-the-art database systems assume that there is always enough workload knowledge and idle time to tune the system with the proper indices, with the proper statistics and with any other data structure which is expected to speed up data access. With big data arriving quickly, unpredictably and with the need to react fast, we do not have the luxury to spend considerable amounts of time in tuning anymore. Second, database systems are designed with the main assumption that we should always consume all data in an effort to provide a correct and complete answer. As the data grows bigger, this becomes a significantly more expensive task.

Overall, before being able to use a database system for posing queries, we first need to go through a complex and time consuming installation process to (a) load data inside the database system and (b) to tune the system. These steps require not only a significant amount of time (i.e., in the order of several hours for a decent database size), but also they require expert knowledge as well as workload knowledge. In other words, we need to know exactly what kind of queries we are going to pose such as we can tune the system accordingly. However, when we are in need to explore a big data pile, then we do not necessarily know exactly what kind of queries we would like to pose before the exploration process actually progresses; the answer to one query leads to the formulation of the next query.

Attempts to “throw more silicon” to the problem, i.e., with big data clusters, can allow for more scalability (until the data grows even bigger) but at the expense of wasted resources when consuming data which is not really necessary for the exploration path. This brings yet another critical side effect of big data into the picture, i.e., energy consumption. Overall, high performance computing and exploitation of large clusters are complementary to the approaches described in this chapter; to deal with big data we need innovations at all fronts.

Because more is Different. We cannot use past solutions to solve radically new problems. The main observation is that with more data, the query-processing paradigm has to change as well. Processing all data is not possible; in fact, often it is not even necessary. For example, a scientist in the astronomy domain is interested in studying parts of the sky at a time searching for interesting patterns, maybe even looking for specific properties at a time. This means that the numerous Terabytes of data brought every few hours by modern telescopes are not relevant all the time. Why should a scientist spend several hours loading all data in a database? Why should they spend several hours indexing all the data? Which data parts are of importance becomes apparent only after going over parts of the data and at least after partially understudying the trends. To make things worse; in a few hours several more Terabytes of data will arrive, i.e., before we make sense of the previous batch of data.

Similarly, in a business analytics setting, changing the processing paradigm can be of critical importance. As it stands, now analysts or tools need to scan all data in search for interesting patterns. Yet, in many emerging applications there is no slack time to waste; answers are needed fast, e.g., when trying to

figure out user behavior or news trends, when observing traffic behavior or network monitoring for fraud detection.

Data Exploration

With such overwhelming amounts of data, *data exploration* is becoming a new and necessary paradigm of query processing, i.e., when we are in search for interesting patterns often not knowing a priori exactly what we are looking for. For example, an astronomer wants to browse parts of the sky to look for interesting effects, while a data analyst of an IT business browses daily data of monitoring streams to figure out user behavior patterns. What both cases have in common is a daily stream of big data, i.e., in the order of multiple Terabytes and the need to observe “something interesting and useful”.

Next generation database systems should interpret queries by their intent, rather than as a contract carved in stone for complete and correct answers. The result in a user query should aid the user in understanding the database’s content and provide guidance to continue the data exploration journey. Data analysts should be able to stepwise explore deeper and deeper the database, and stop when the result content and quality reaches their satisfaction point. At the same time, response times should be close to instant such that they allow users to interact with the system and explore the data in a contextualized way as soon as data become available.

With systems that support data exploration we can immediately discard the main bottleneck that stops us from consuming big data today; instead of considering a big data set in one go with a slow process, exploration-based systems can incrementally and adaptively guide users towards the path that their queries and the result lead. This helps us avoid major inherent costs, which are directly affected by the amount of data input and thus are showstoppers nowadays. These costs include numerous procedures, steps and algorithms spread throughout the whole design of modern data management systems.

Key Goals: Fast, Interactive, and Adaptive. For efficient data exploration to work, there are few essential goals.

First the system should be fast to the degree that it feels interactive, i.e., the user poses a question and a few seconds later an answer appears. Any data that we load does not have to be complete. Any data structure that we built does not have to represent all data or all value ranges. The answer itself does not have to represent a correct and complete result but rather a hint of how the data looks like and how to proceed further, i.e., what the next query should be. This is essential in order to engage data analysts in a seamless way; the system is not the bottleneck anymore.

Second, the system and the whole query processing procedure should be adaptive in the sense that it adapts to the user requests; it proceeds with actions that speed up the search towards eventually getting the full answer

the user is looking for. This is crucial in order to be able to finally satisfy the user needs after having sufficiently explored the data.

Metaphor Example. The observations to be made about the data in this case resemble an initially empty picture; the user sees one pixel at a time with every query they pose to the system. The system makes sure it remembers all pixels in order to guide the user towards areas of the picture where interesting shapes start to appear. Not all the picture has to be completed for interesting effects to be seen from a high level point of view, while again not all the picture is needed for certain areas to be completed and seen in more detail.

Data Exploration Techniques. In the rest of this chapter, we discuss a string of novel data exploration techniques that aim to rethink database architectures with big data in mind. We discuss (a) adaptive indexing to build indexes on-the-fly as opposed to a priori, (b) adaptive loading to allow for direct access on raw data without a priori loading steps and (c) database architectures for approximate query processing to work over dynamic samples of data.

Adaptive Indexing

In this section, we present adaptive indexing. We discuss the motivation for adaptive indexing in dynamic big data environments as well as the main bottlenecks of traditional indexing approaches. This section gives a broad description of the state-of-the-art in adaptive indexing, including topics such as updates, concurrency control and robustness.

Indexing. Good performance in state-of-the-art database systems relies largely on proper tuning and physical design, i.e., creating the proper accelerator structures, called indices. Indices are exploited at query processing time to provide fast data access. Choosing the proper indices is a major performance parameter in database systems; a query may be several orders of magnitude faster if the proper index is available and is used properly. The main problem is that the set of potential indices is too large to be covered by default. As such, we need to choose a subset of the possible indices and implement only those.

In the past, the choice of the proper index collection was assigned to database administrators (DBAs). However, as applications became more and more complex index selection became too complex for human administration alone. Today, all modern database systems ship with tuning advisor tools. Essentially those tools provide suggestions regarding which indices should be created. A human database administrator is then responsible of making and implementing the final choices.

Offline Indexing. The predominant approach is offline indexing. With offline indexing, all tuning choices happen up front, assuming sufficient workload knowledge and idle time. Workload knowledge is necessary in order to

determine the appropriate tuning actions, i.e., to decide which indices should be created, while idle time is required in order to actually perform those actions. In other words, we need to know what kind of queries we are going to ask and we need to have enough time to prepare the system for those queries.

Big Data Indexing Problems. However, in dynamic environments with big data, workload knowledge and idle time are scarce resources. For example, in scientific databases, new data arrive on a daily or even hourly basis, while query patterns follow an exploratory path as the scientists try to interpret the data and understand the patterns observed; there is no time and knowledge to analyze and prepare a different physical design every hour or even every day; even a single index may take several hours to create.

Traditional indexing presents three fundamental weaknesses in such cases: (a) the workload may have changed by the time we finish tuning; (b) there may be no time to finish tuning properly; and (c) there is no indexing support during tuning.

Database Cracking. Recently, a new approach, called database cracking, was introduced to the physical design problem. Cracking introduces the notion of continuous, incremental, partial and on demand adaptive indexing. Thereby, indices are incrementally built and refined during query processing. The net effect is that there is no need for any upfront tuning steps. In turn, there is no need for any workload knowledge and idle time to set up the database system. Instead, the system autonomously builds indices during query processing, adjusting fully to the needs of the users. For example, as a scientist starts exploring a big data set, query after query, the system follows the exploration path of the scientist, incrementally building and refining indices only for the data areas that seem interesting for the exploration path. After a few queries, performance adaptively improves to the level of a fully tuned system. From a technical point of view cracking relies on continuously physically reorganizing data as users pose more and more queries.

Every query is treated as a hint on how data should be stored.

Column-stores. Before we discuss cracking in more detail, we give a short introduction to column-store databases. Database cracking was primarily designed for modern column-stores and thus it relies on a number of modern column-store characteristics. Column-stores store data one column at a time in fixed-width dense arrays. This representation is the same both for disk and for main-memory. The net effect compared to traditional row-stores is that during query processing, a column-store may access only the referenced data/columns. Similarly, column-stores rely on bulk and vector-wised processing. Thus, a select operator typically processes a single column in one go or in a few steps, instead of consuming full tuples one-at-a-time. Specifically for database cracking the column-store design allows for efficient physical reorganization of arrays. In effect, cracking performs all physical reorganization actions efficiently in one go over a single column; it does not have to touch other columns.

Selection Cracking Example. We now briefly recap the first adaptive indexing technique, selection cracking, as it was introduced in (Idreos, Kersten and Manegold, Database Cracking 2007). The main innovation is that the physical data store is continuously changing with each incoming query q, using q as a hint on how data should be stored. Assume an attribute A stored as a fixed-width dense array in a column-store. Say a query requests all values where A<10. In response, a cracking DBMS clusters all tuples of A with A<10 at the beginning of the respective column C, while pushing all tuples with A>=10 to the end. In other words, it partitions on-the-fly and in-place column C using the predicate of the query as a pivot. A subsequent query requesting A>=v1 where v1>=10, has to search and crack only the last part of C where values A>=10 reside. Likewise, a query that requests A<v2, where v2<10, searches and cracks only the first part of C. All crack actions happen as part of the query operators, requiring no external administration.

The terminology ``cracking" reflects the fact that the database is partitioned (cracked) into smaller and manageable pieces.

Data Structures. The cracked data for each attribute of a relational table are stored in a normal column (array). The very first query on a column copies the base column to an auxiliary column where all cracking happens. This step is used such as we can always retrieve the base data in its original form and order. In addition, cracking uses an AVL-tree to maintain partitioning information such as which pieces have been created, which values have been used as pivots, etc.

Continuous Adaptation. The cracking actions continue with every query. In this way, the system reacts to every single query, trying to adjust the physical storage, continuously reorganizing columns to fit the workload patterns. As we process more queries, the more performance improves. In essence, more queries introduce more partitioning, while pieces become smaller and smaller. Every range query or more precisely every range select operator needs to touch at most two pieces of a column, i.e., those pieces that appear at the boundaries of the needed value range. With smaller pieces, future queries need less effort to perform the cracking steps and as such performance gradually improves.

To avoid the extreme status where a column is completely sorted, cracking poses a threshold where it stops cracking a column for pieces which are smaller than L1 cache. There are two reasons for this choice. First, the AVL-tree, which maintains the partitioning information, grows significantly and causes random access when searching. Second, the benefit brought by cracking pieces that are already rather small is minimal. As such, if during a query, a piece smaller than L1 is indicated for cracking, the system completely sorts this piece with an in-memory quick sort. The fact that this piece is sorted is marked in the AVL-tree. This way, any future queries, for which the bounds of the requested range fall within a sorted piece, can simply binary search for their target bound.

Performance Examples. In experiments with the Skyserver real query and data logs, a database system with cracking enabled, finished answering 160.000 queries, while a traditional system was still half way creating the proper indices and without having answered a single query (Halim, et al. 2012). Similarly, in experiments with the business standard TPC-H benchmark, perfectly preparing a system with all the proper indices took ~3 hours, while a cracking database system could answer all queries in a matter of a few seconds with zero preparation, while still reaching optimal performance, similar to that of the fully indexed system (Idreos, Kersten and Manegold, Self-organizing Tuple Reconstruction In Column-stores 2009).

Being able to provide this instant access to data, i.e., without any tuning, while at the same time being able to quickly, adaptively and incrementally approach optimal performance levels in terms of response times, is exactly the property which creates a promising path for data exploration. The rest of the chapter discusses several database architecture challenges that arise when trying to design database kernels where adaptive indexing becomes a first class citizen.

Sideways Cracking. Column-store systems access one column at a time. They rely on the fact that all columns of the same table are aligned. This means that for each column, the value in the first position belongs in the first tuple, the one in the second position belongs in the second tuple and so on. This allows for efficient query processing for queries which request multiple columns of the same table, i.e., for efficient tuple reconstruction.

When cracking physically reorganizes one column, the rest of the columns of the same table remain intact; they are separate physical arrays As a result, with cracking, columns of the same table are not aligned anymore. Thus, when a future query needs to touch more than one columns of the same table, then the system is forced to perform random access in order to reconstruct tuples on-the-fly. For example, assume a selection on a column A, followed by a projection on another column B of the same table. If column A has been cracked in the past, then the tuple IDs, which is the intermediate result out of the select operator on A, are in a random order and lead to an expensive access to fetch the qualifying values from column B.

One approach could be that every time we crack one column, we also crack in the same way all columns of the same table. However, this defeats the purpose of exploiting column-stores; it would mean that every single query would have to touch all attributes of the referenced table as opposed to only touching the attributes which are truly necessary for the current query.

Sideways cracking solves this problem by working on pairs of columns at a time (Idreos, Kersten and Manegold, Self-organizing Tuple Reconstruction In Column-stores 2009) and by adaptively forwarding cracking actions across the columns of the same table. That is for a pair of columns A and B, during the cracking steps on A, the B values follow this reorganization. The values of A and B are stored together in a binary column format, making the physical

reorganization efficient. Attribute A is the head of this column pair, while attribute B is the tail. When more than two columns are used in a query, sideways cracking uses bit vectors to filter intermediate results while working across multiple column-pairs of the same head. For example, in order to do a selection on attribute A and two aggregations, one on attribute B and one attribute C, sideways cracking uses pairs AB and AC. Once both pairs are cracked in the same way using the predicates on A, then they are fully aligned and they can be used in the same plans without tuple reconstruction actions.

Essentially, sideways cracking performs tuple reconstructions via incremental cracking and alignment actions as opposed to joins. For each pair, there is a log to maintain the cracking actions that have taken place in this pair as well as in other pairs that use the same head attribute. Two column-pairs of the same head are aligned when they have exactly the same history, i.e., they have been cracked for the same bounds and in exactly the same order.

Partial Cracking. The pairs of columns created by sideways cracking can result in a large set of auxiliary cracking data. With big data this is an important concern. Cracking creates those column pairs dynamically, i.e., only what is needed is created and only when it is needed. Still though, the storage overhead may be significant. Partial cracking solves this problem by introducing partial cracking columns (Idreos, Kersten and Manegold, Self-organizing Tuple Reconstruction In Column-stores 2009). With partial cracking, we do not need to materialize complete columns; only the values needed by the current hot workload set are materialized in cracking columns. If missing values are requested by future queries, then the missing values are fetched from the base columns the first time they are requested.

With partial cracking, a single cracking column becomes a logical view of numerous smaller physical columns. In turn, each one of the small columns, is cracked and accessed in the same way as described for the original database cracking technique, i.e., it is continuously physically reorganized as we pose queries.

Users may pose a storage budget and cracking makes sure it will stay within the budget by continuously monitoring the access patterns of the various materialized cracking columns. Each small physical column of a single logical column is completely independent and can be thrown away and recreated at any time. For each column, cracking knows how many times it has been accessed by queries and it uses an LRU policy to throw away columns when space for a new one is needed.

Updates. Updates pose a challenge since they cause physical changes to the data which in combination with the physical changes caused by cracking may lead to significant complexity. The solution proposed in (Idreos, Kersten and Manegold, Updating a Cracked Database 2007) deals with updates by deferring update actions for when relevant queries arrive. In the same spirit as with the rest of the cracking techniques, cracking updates do not do any work until it is unavoidable, i.e., until a query, which is affected by a pending update, arrives. In this way, when an update comes, it is simply put aside. For

each column, there is an auxiliary delete column where all pending deletes are placed and an auxiliary insertions column where all pending inserts are placed. Actual updates are a combination of a delete and then an insert action.

Each query needs to check the pending deletes and inserts for pending actions that may affect it. If there are any, then those qualifying pending insertions and deletions are merged with the cracking columns on-the-fly. The algorithm for merging pending updates into cracking columns takes advantage of the fact that there is no strict order within a cracking column. For example, each piece in a cracking column contains values within a given value range but once we know that a new insertion for example should go within this piece, then we can place it in any position of the piece; within each cracking piece there is no strict requirement for maintaining any order.

Adaptive Merging. Cracking can be seen as an incremental quicksort where the pivots are defined by the query predicates. Adaptive merging was introduced as a complementary technique, which can be seen as an incremental merge sort where the merging actions are defined by the query predicates (Graefe and Kuno, Self-selecting, self-tuning, incrementally optimized indexes 2010). The motivation is mainly towards disk-based environments and towards providing fast convergence to optimal performance.

The main design point of adaptive merging is that data is horizontally partitioned into runs. Each run is sorted in memory with a quicksort action. This preparation step is done with the first query and results to an initial column that contains the various runs. From there on, as more queries arrive data is moved from the initial column to a results column where the final index is shaped. Every query merges into the results column only data which are defined by its selection predicates and which are missing from the results column. If a query is covered fully by the results column, then it does not need to touch the initial runs. Data that is merged is immediately sorted in place in the results column; once all data is merged the results column is actually a fully sorted column. With data pieces being sorted both in the initial column and in the results column, queries can exploit binary search both during merging and when accessing only the results column.

Hybrids. Adaptive merging improves over plain cracking when it comes to convergence speed, i.e., the number of queries needed to reach performance levels similar to that of a full index is significantly reduced. This behavior is mainly due to the aggressive sorting actions during the initial phase of adaptive merging; it allows future queries to access data faster. However, these sorting actions put a sizeable overhead on the initial phase of a workload, causing the very first query to be significantly slower. Cracking, on the other hand, has a much more smooth behavior, making it more lightweight to individual queries. However, cracking takes much longer to reach the optimal index status (unless there is significant skew in the workload).

The study in (Idreos, Manegold, et al. 2011) presents these issues and proposes a series of techniques that blend the best properties of adaptive merging with the best properties of database cracking. A series of hybrid algorithms are proposed where one can tune how much initialization overhead and how much convergence speed is needed. For example, the crack-crack hybrid (HCC) uses the same overall architecture as adaptive merging, i.e., using an initial column and a results column where data is merged based on query predicates. However, the initial runs are now not sorted; instead, they are cracked based on query predicates. As a result the first query is not penalized as with adaptive merging. At the same time, the data placed in the results column is not sorted in place. Several combinations are proposed where one can crack, sort or radix cluster the initial column and the result column. The crack-sort hybrid, which cracks the initial column, while it sorts the pieces in the result column, brings the best overall balance between initialization and converge costs (Idreos, Manegold, et al. 2011).

Robustness. Since cracking reacts to queries, its adaptation speed and patterns depend on the kind of queries that arrive. In fact, cracking performance crucially depends on the arrival order of queries. That is, we may run exactly the set of queries twice in slightly different order and the result may be significantly different in terms of response times even though exactly the same cracking index will be created. To make this point more clear consider the following example. Assume a column of 100 unique integers in [0,99]. Assume a first query that asks for all values v where v<1. As a result, cracking partitions the column into two pieces. In piece P1 we have all values in [0,1) and in piece P2 we have all values in [1,99]. The net effect is that the second piece still contains 99 values, meaning that the partitioning achieved by the first query is not so useful; any query falling within the second piece still has to analyze almost all values of the column. Now assume that the second query requests all values v where v<2. Then, the third query requests all values v where v<3 and so on. This sequence results in cracking having to continuously analyze large portions of the column as it always leaves back big pieces. The net effect is that convergence speed is too slow and in the worst case cracking degrades to a performance similar to that of a plain scan for several queries, resulting in a performance which is not robust (Halim, et al. 2012).

To solve the above problem, (Halim, et al. 2012) proposes stochastic cracking. The main intuition is that stochastic cracking plugs in stochastic cracking actions during the normal cracking actions that happen during processing. For example, when cracking a piece of a column for a pivot X, stochastic cracking adds an additional cracking step where this piece is also cracked for a pivot which is randomly chosen. As a result the chances of leaving back big uncracked pieces becomes significantly smaller.

Concurrency Control. Cracking is based on continuous physical reorganization of the data. Every single query might have side effects. This is in strong contrast with what normally happens in database systems where plain queries do not have side effects on the data. Not having any side effects

means that read queries may be scheduled to run in parallel. Database systems heavily rely on this parallelism to provide good performance when multiple users access the system simultaneously. On the other hand, with cracking, every query might change the way data is organized and as a result it is not safe to have multiple queries working and changing the same data in parallel.

However, we would like to have both the adaptive behavior of database cracking, while still allowing multiple users to query big data simultaneously. The main trick to achieve this is to allow concurrent access on the various pieces of each cracking column; two different queries may be physically reorganizing the same column as long as they do not touch the exact same piece simultaneously (Graefe, Halim, et al. 2012). In this way, each query may lock a single piece of a cracking column at a time, while other queries may be working on the other pieces. As we create more and more pieces there are more opportunities to increase the ability for multiple queries to work in parallel. This bonds well with the adaptive behavior of database cracking; if a data area becomes hot, then more queries will arrive to crack it into multiple pieces and subsequently more queries will be able to run in parallel because more pieces exist.

Contrary to concurrency control for typical database updates, with adaptive indexing during read queries we only change the data organization; the data contents remain intact. For this reason, all concurrency mechanisms for adaptive indexing may rely on latching as opposed to full-fledged database locks, resulting in a very lightweight design (Graefe, Halim, et al. 2012).

Summary. Overall, database cracking opens an exciting path towards database systems that inherently support adaptive indexing. By not requiring any workload knowledge and any tuning steps, we can significantly reduce the time it takes to query newly arrived data, assisting data exploration.

Adaptive Loading

The previous section described the idea of building database kernels that inherently provide adaptive indexing capabilities. Indexing is one of the major bottlenecks when setting up a database system; but it is not the only one. In this section, we focus on another crucial bottleneck, i.e., on data loading. We discuss the novel direction of adaptive loading to enable database systems to bypass the loading overhead and immediately be able to query data before even being loaded in a database.

The Loading Bottleneck. Data loading is a necessary step when setting up a database system. Essentially, data loading copies all data inside the database system. From this point on, the database fully controls the data; it stores data in its own format and uses its own algorithms to update and access the data. Users cannot control the data directly anymore; only through the database system. The reason to perform the loading step is to enable good performance during query processing; by having full control on the data, the

database system can optimize and prepare for future data accesses. However, the cost of copying and transforming all data is significant; it may take several hours to load a decent data size even with parallel loading.

As a result, in order to use the sophisticated features of a database system, users have to wait until their data is loaded (and then tuned). However, with big data arriving at high rates, it is not feasible anymore to reserve several hours for data loading as it creates a big gap between data creation and data exploitation.

External Files. One feature that almost all open source and commercial database products provide is external tables. External files are typically in the form of raw text-based files in CSV format (comma-separated values). With the external tables functionality one can simply attach a raw file to a database without loading the respective data. When a query arrives for this file, the database system dynamically goes back to the raw file to access and fetch the data on-the-fly. This is a useful feature in order to delay data loading actions but unfortunately it is not a functionality that can be used for query processing. The reason is that it is too expensive to query raw files; there are several additional costs involved. In particular, parsing and tokenizing costs dominate the total query processing costs. Parsing and tokenizing are necessary in order to distinguish the attribute values inside raw files and to transform them into binary form. For this reason, the external tables functionality is not being used for query processing.

Adaptive Loading. The NoDB project recently proposed the adaptive loading direction (Idreos, Alagiannis, et al. 2011) (Alagiannis, et al. 2012); the main idea is that loading actions happen adaptively and incrementally during query processing and driven by the actual query needs. Initially, no loading actions take place; this means that there is no loading cost and that users can immediately query their data. With every query the system adaptively fetches any needed data from the raw data files. At any given time, only data needed by the queries is loaded. The main challenge of the adaptive loading direction is to minimize the cost to touch the raw data files during query processing, i.e., to eliminate the reason that makes the external tables fuctionality unusable for querying.

The main idea is that as we process more and more queries, NoDB can collect knowledge about the raw files and significantly reduce the data access costs. For example, it learns about how data resides on raw files in order to better look for it, if needed, in the future.

Selective Parsing. NoDB pushes selections down to the raw files in order to minimize the parsing costs. Assume a query that needs to have several filtering conditions checked for every single row of a data file. In a typical external files process, the system tokenizes and parses all attributes in each row of the file. Then, it feeds the data to the typical data flow inside the database system to process the query. This incurs a maximum parsing and tokenizing cost. NoDB removes this overhead by performing parsing and tokenizing selectively on a row-by-row basis, while applying the filtering

predicates directly on the raw file. The net benefit is that as soon any of the filtering predicates fails, then NoDB can abandon the current row and continue with the next one, effectively avoiding significant parsing and tokenizing costs. To achieve all these step, NoDB overloads the scan operator with the ability to access raw file in addition to loaded data.

Indexing. In addition, during parsing, NoDB creates and maintains an index to mark positions on top of the raw file. This index is called positional map and its functionality is to provide future queries with direct access to a location of the file that is close to what they need. For example, if for a given row we know the position of the 5^{th} attribute and the current query needs to analyze the 7^{th} attribute, then the query only needs to start parsing as of the attribute on the 5^{th} position of the file. Of course, given that we cannot realistically assume fixed length attributes, the positional map needs to maintain information on a row-by-row basis. Still though, the cost is kept low, as only a small portion of a raw file needs to be indexed. For example, experiments in (Alagiannis, et al. 2012) indicate that once 15% of a raw file is indexed, then performance reaches optimal levels.

Caching. The data fetched from the raw file is adaptively cached and reused if similar queries arrive in the future. This allows the hot workload set to always be cached and the need to fetch raw data appears only during workload shifts. The policy used for cache replacement is LRU in combination with adaptive loading specific parameters. For example, integer attributes have a priority over string attributes in the cache; fetching string attributes back from the raw file during future queries is significantly less expensive than fetching integer attributes. This is because the parsing costs for string attributes are very low compared to those for integer values.

Statistics. In addition, NoDB creates statistics on-the-fly during parsing. Without proper statistics, optimizers cannot make good choices about query plans. With adaptive loading, the system is initiated without statistics as no data is loaded up front. To avoid bad plans and to guarantee robustness, NoDB immediately calculates statistics the very first time an attribute of a given raw file is requested by a query. This puts a small overhead at query time, but it allows us to avoid bad optimization choices.

Splitting Files. When accessing raw files, we are limited in exploiting the format of the raw files. Typically, data is stored in CSV files where each row represents an entry in a relational table and each file represents all data in a single relational table. As a result, every single query that needs to fetch data from raw files has to touch all data. Even with selective parsing and indexing, at the low level the system still needs to touch almost all the raw file. If the data was a priori loaded and stored in a column-store format, then a query would need to touch only the data columns it really needs. NoDB proposed the idea of text cracking, where during parsing the raw file is separated into multiple files and each file may contain one or more of the attributes of the original raw file (Idreos, Alagiannis, et al. 2011). This process works recursively and as a result future queries on the raw file, can significantly

reduce the amount of data they need to touch by having to work only on smaller raw files.

Data Vaults. One area where adaptive loading can have a major impact is sciences. In the case of scientific data management, several specialized formats already exist and are in use for several decades. These formats store data in a binary form and often provide indexing information, e.g., in the form of clustering data based on date of creation. In order to exploit database systems for scientific data management, we would need to transform data from the scientific format into the database format, incurring a significant cost. The data vaults project provides a two-level architecture that allows exploiting the metadata in scientific data formats for adaptive loading operations (Ivanova, Kersten and Manegold 2012). Given that the scientific data is already in binary format, there are no considerations regarding parsing and tokenizing costs. During the initialization phase, data vaults load only the metadata information, resulting in a minimal set up cost. During query processing time, the system uses the metadata to guide the queries to the proper files and to transform only the needed data on-the-fly. This way, without performing any a priori transformation of the scientific data, we can pose queries through the database system directly and selectively.

Summary. Loading represents a significant bottleneck; it raises a wall between users and big data. Adaptive loading directions provide a promising research path towards systems that can be usable immediately as soon as data arrive by removing loading costs.

Sampling-based Query Processing

Loading and indexing are the two essential bottlenecks when setting up a database system. However, even after all installation steps are performed, there are more bottlenecks to deal with; this time bottlenecks appear during query processing. In particular, the requirements for correctness and completeness raise a significant overhead; every single query is treated by a database system as a request to find all possible and correct answers.

This inherit requirement for correctness and completeness has its roots in the early applications of database systems, i.e., mainly in critical sectors such as in banking and financial applications where errors cannot be tolerated. However, with modern big data applications and with the need to explore data, we can afford to sacrifice correctness and completeness in favor of improved response times. A query session which may consist of several exploratory queries can lead in exactly the same result, regardless of whether the full answer is returned every time; in an exploratory session users are mainly looking for hints on what the next query should be and a partial answer may already be informative enough.

In this section, we discuss a number of recent approaches to create database systems that are tailored for querying with partial answers, sacrificing correctness and completeness for improved response times.

Sciborg. Sciborg proposed the idea of working over data that is organized in a hierarchy of samples (Sidirourgos, Kersten and Boncz 2011). The main idea is that queries can be performed over a sample of the data providing a quick response time. Subsequently, the user may choose to ask for more detail and to query more samples. Essentially, this is a promising research path to enable interactive query processing. The main innovation in Sciborg is that samples of data are not simply random samples; instead, Sciborg creates weighted samples driven by past query processing actions and based on the properties of the data. In this way, it can better follow the needs of the users by collecting relevant data together such as users can infer interesting patterns using only a small number of samples.

Blink. Another recent project, Blink, proposes a system where data is also organized in multiple samples (Agarwal, et al. 2012). The characteristic of Blink is its seamless integration with cloud technology, being able to scale to massive amounts of data and processing nodes.

Both the Blink and the Sciborg projects represent a vision to create database kernels where the system inherently supports query processing over samples. For example, the user does not have to create a sample explicitly and then query it, followed by the creation of a different sample, while repeating this process multiple times. In a database architecture that supports samples at its core, this whole process is transparent to the user and has the potential to be much more effective. For example, with tailored database kernels (a) the samples are created with minimal storage overhead, (b) they adapt continuously and (c) query results over multiple samples can be merged dynamically by the system.

One-minute DB Kernels. Another vision in the direction of exploration-based database kernels is the one-minute database kernels idea (Kersten, et al. 2011). Similar to Sciborg and Blink, the main intuition is that correctness and completeness are sacrificed in favor of performance; however, contrary to past approaches, this happens at a very low level, i.e., at the level of database operators. Every decision in the design of database algorithms can be reconsidered to avoid expensive actions by sacrificing correctness. For example, a join operator may choose to drop data from the inner join input as soon as the size of the hash table exceeds the size of the main memory or even the size of CPU cache. A smaller hash table is much faster to create and it is also much faster to probe, avoiding cache misses. Similar decisions can be made across the whole design of database kernels.

Essentially, the one-minute database kernels approach is equivalent to the sample-based ideas. The difference is that it pushes the problem at a much lower level where possibly we may have better control of parameters that affect performance. One of the main challenges is to be able to provide quality guarantees for the query results.

dbTouch. One significant bottleneck when querying database systems is the need to be an expert user; one needs to be aware of the database schema and needs to be fluent in SQL. When it comes to big data exploration, we would like to make data accessible to more people and to make the whole process of discovering interesting patterns as easy as possible. dbTouch extends the vision of sample-based processing with the notion of creating database kernels which are tailored for touch-based exploration (Idreos and Liarou, dbTouch: Analytics at your Fingetips 2013). Data appears in a touch device in a visual form, while users can simply touch the data to query. For example, a relational table may be represented as a table shape and a user may slide a finger over the table to run a number of aggregations. dbTouch is not about formulating queries; instead it proposes a new database kernel design which reacts instantly to touch. Users do not pose queries as in normal systems; in dbTouch users point to interesting data and the system continuously reacts to every touch. Every touch corresponds to analyzing a single tuple are a few tuples, while a slide gesture captures multiple tuples. As such, only a sample of the data is processed every time, while now the user has full control regarding which data is processed and when; by changing the direction or the speed of a slide gesture, users can control the exploration process, while observing running results as they are visualized by dbTouch.

The main challenge with dbTouch is in designing database kernels that can react instantly to every touch and to provide quick response times even though the database does not control anymore the order and the kind of data processed for every query session.

Summary. Overall, correctness and completeness pose a significant bottleneck during query time; with big data this problem becomes a major showstopper as it becomes extremely expensive to consume big piles of data. The novel research directions described in this chapter make a first step towards a new era of database kernels where performance becomes more important than correctness and where exploration is the main query processing paradigm.

Summary

In the presence of big data, query processing is facing significant new challenges. A particular aspect of those challenges has to do with the fact that there is not enough time and workload knowledge to properly prepare and tune database management systems. In addition, producing correct and complete answers by consuming all data within reasonable time bounds is becoming harder and harder. In this chapter, we discussed the research direction of data exploration where adaptive and incremental processing become first class citizens in database architectures.

Adaptive indexing, adaptive loading and sampling-based database kernels provide a promising path towards creating dedicated exploration systems. It represents a widely open research area as we need to reconsider every single aspect of database design established in the past.

Bibliography

Abouzeid, A., K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. "HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads." *Proceedings of the Very Large Databases Endowment (PVLDB)* 2, no. 1 (2009): 922-933.

Agarwal, Sameer, Aurojit Panda, Barzan Mozafari, Anand P. Iyer, Samuel Madden, and Ion Stoica. "Blink and It's Done: Interactive Queries on Very Large Data." *Proceedings of the Very Large Databases Endowment (PVLDB)* 5, no. 6 (2012): 1902-1905.

Alagiannis, Ioannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. "NoDB: efficient query execution on raw data files." *ACM SIGMOD International Conference on Management of Data.* 2012.

Boncz, Peter A., Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution." *Biennial Conference on Innovative Data Systems Research (CIDR).* 2005. 225-237.

Dean, J., and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." *USENIX Symposium on Operating Systems Design and Implementation (OSDI).* 2004. 137- 150.

Graefe, Goetz, Felix Halim, Stratos Idreos, and Stefan Manegold Harumi A. Kuno. "Concurrency Control for Adaptive Indexing." *Proceedings of the Very Large Databases Endowment (PVLDB)* 5, no. 7 (2012): 656-667.

Graefe, Goetz, and Harumi A. Kuno. "Self-selecting, self-tuning, incrementally optimized indexes." *International Conference on Extending Database Technology (EDBT).* 2010.

Hadapt. 2012. http://www.hadapt.com/.

Halim, Felix, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. "Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores." *Proceedings of the Very Large Databases Endowment (PVLDB)* 5, no. 6 (2012): 502-513.

Idreos, Stratos, and Erietta Liarou. "dbTouch: Analytics at your Fingetips." *International Conference on Innovative Data Systems Research (CIDR).* 2013.

Idreos, Stratos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. "Here are my Data Files. Here are my Queries. Where are my Results?"

International Conference on Innovative Data Systems Research (CIDR),. 2011.

Idreos, Stratos, Martin Kersten, and Stefan Manegold. "Database Cracking." *International Conference on Innovative Data Systems Research (CIDR).* 2007.

—. "Self-organizing Tuple Reconstruction In Column-stores." *ACM SIGMOD International Conference on Management of Data.* 2009.

—. "Updating a Cracked Database ." *ACM SIGMOD International Conference on Management of Data.* 2007.

Idreos, Stratos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. "Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores." *Proceedings of the Very Large Databases Endowment (PVLDB)* 4, no. 9 (2011): 585-597.

Ivanova, Milena, Martin L. Kersten, and Stefan Manegold. "Data Vaults: A Symbiosis between Database Technology and Scientific File Repositories." *International Conference on Scientific and Statistical Database Management (SSDBM).* 2012.

Kersten, Martin, Stratos Idreos, Stefan Manegold, and Erietta Liarou. "The Researcher's Guide to the Data Deluge: Querying a Scientific Database in Just a Few Seconds." *Proceedings of the Very Large Databases Endowment (PVLDB)* 4, no. 12 (2011): 174-177.

Platfora. 2012. http://www.platfora.com/.

Sidirourgos, Lefteris, Martin L. Kersten, and Peter A. Boncz. "SciBORQ: Scientific data management with Bounds On Runtime and Quality." *International Conference on Innovative Data systems Research (CIDR).* 2011.

Stonebraker, Michael, et al. "C-Store: A Column-oriented DBMS." *International Conference on Very Large Databases (VLDB).* 2005. 553-564.