University of Kaiserslautern
Department of Computer Science
Databases and Information Systems Group

Seminar "Recent Trends in Database Research"

Query processing on raw files

Author:
Vítor Uwe Reus

Supervisor:
M.Sc. Caetano Sauer

July 8, 2013

Table of Contents

Query processing on raw files

Vítor Uwe Reus

Seminar "*Recent Trends in Database Research*", Summer 2013
Databases and Information Systems Group
University of Kaiserslautern
reus@cs.uni-kl.de

Abstract. This work aims to investigate the state of the art of query processing over raw files. As data volume grows in several areas, more and more raw files are being used instead of the classical DBMS store approach. There are several reasons for preferring raw files instead of a DBMS such as speed, file formats or schema-less data. Some implementations exists to provide this kind of querying, going from the most basic ones, like simply loading the data into a DBMS, to more complex ones that actually uses raw files as the storage layer. Each of the solutions have advantages and disadvantages. Current solutions can be classified depending its load and query time. Hybrid solutions and how close they are to the ideal are presented. At least, some conclusions and insights about the area are presented.

1 Introduction

Raw files are one of many ways of storing computer data. They contain minimally processed information from a data source. Raw files are sometimes presented in a human-readable format, in contrast to the binary format used in traditional DBMS. Traditional DBMS usually store the information in some optimized binary format such as B-trees to accelerate query response time. For this reason, querying raw files directly is not a viable option if low response times are required. However, there are some situations where it might be useful to use raw files instead of a RDBMS.

There is a trend to use raw files when there are huge amounts of data because it might take too much time to load existing huge files into a traditional DBMS. When there is the necessity of fast information extraction from a huge amount of data, there might not be enough time to write loading scripts, schema, which may be unknown a priori, and to wait all the data to be transferred into the DBMS, for only then being able to query it and extract the information.

Science has unlocked the 4th paradigm [7] of scientific discovery, which is driven by analysis of huge amounts of data. The source of the data can be from simulations or observations. It is usually a stream of data, which in an experimental scientific environment is more convenient to simple append to a raw file than to set up a database and insert that information. Therefore, it is not uncommon that scientists ends up with huge raw files that needs to be processed to extract information.

Industry established the term Big Data to refer to business models which deal with big amounts of data, like Internet services that serve billions of users. This kind of data size is, like scientific data, too big that makes consuming all of it difficult.

Raw files may also provide better information interoperability than traditional RDBMS. There are several standard file formats like CSV, XML, JSON or YAMS that can be used as the storage layer instead of the proprietary secret format in which a database stores the data. There are two kinds of interoperability that raw files might provide. Information interoperability and application interoperability.

Information interoperability is the ability to use the data "as is" in several applications. Data migration from one system to another might be much easier using standard files and several different system might share the access to the same files to query information, acting like a unified data source repository, as in Figure 1(a). Also if there is a data input stream, it could be easily appended to the file, and all systems would have this data instantly available.

Application interoperability is the ability to use an application as is over several data sources. If there are several data sources of interest, the application can query information over all them if they use a standard format, as in Figure 1(b).

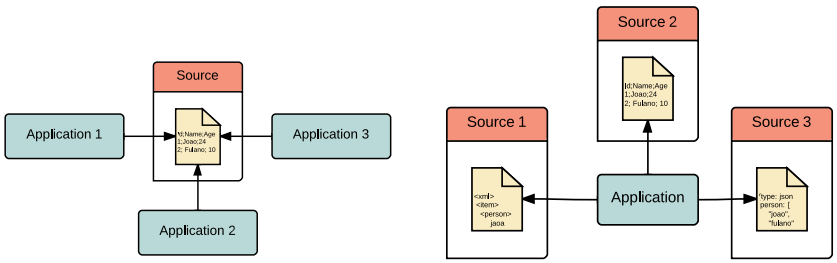

Fig. 1. Interoperability using standard raw files. In 1(a), Information interoperability between several applications. In 1(b), application interoperability through several data sources.

Human-sourced information is another area in which raw files can be useful. Social networks, and Web 2.0 applications are a big source of huge amounts of unstructured, schema-less information. Traditional DBMS are made to be used with structured information, they provide indexes and auxiliary structures that relies on the data's schema. Therefore, traditional DBMS does not offer much advantage for storing this kind of information. Thus, raw files can also be useful to store unstructured information.

As we can see, traditional DBMS are already solving most of the data storing problems, but there are some cases where it might be interesting to use raw files for storing. Having these raw data files, we want to be able to perform fast queries over them. In the following, existing solutions to query raw files are presented.

1.1 State of the art

Currently there exist two alternatives to query raw files. First, we can query the file directly doing some sequential parsing. Secondly, we can load the data into an existing DBMS and then perform regular queries over it, using all the advantages of these well known systems.

Unix tools like AWK are able to perform simple select queries (without joins) over raw files in linear time [8]. AWK may be a good alternative if the data is intended to be queried only once. Assuming that just loading the data into an DBMS also takes around the same linear time, we can affirm that AWK is able to start answering queries faster than using a DBMS. Avoiding the load time is the main advantage of AWK.

Some DBMS already implement the ability to actually use the raw file as the storage layer and perform queries over it [8, 4]. Oracle has external tables and MySQL has the CSV engine. This storage mode makes the best effort to read and process the file in a pipelined architecture, but their implementations are much slower than native tables [4].

MapReduce is an alternative to query large raw files on a distributed file system. It can be classified as a raw file parsing technique if used in its most simple form. MapReduce is a programming model for processing large data sets in parallel. It comprises a Map function, responsible for filtering and processing individual data, and an aggregation function, Reduce, which performs a summary operation, as shown in figure 2. This is a flexible programming model, highly parallelizable and distributable because of its functional nature.

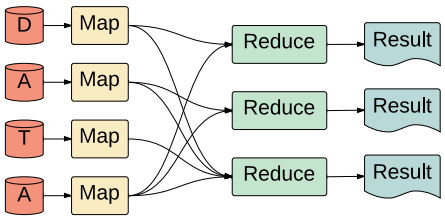

Fig. 2. MapReduce overview.

MapReduce is currently being used to query large raw files. Hadoop is an open-source implementation of the MapReduce model. Writing more complex Hadoop jobs like joins or nested projections is a labor intensive task, compared to simply writing a SQL query. Therefore some efforts like HIVE [11] and PIG

[10] have been made to convert query languages into Hadoop jobs. This languages allow the user to write declarative queries that will be translated into Hadoop jobs. These languages offer a really good abstraction for the Hadoop framework and raw file querying, but they still access raw files in the end. In Section 3, better hybrid MapReduce solutions will be discussed.

The second way to query raw data is to simply load it to an existing DBMS and then query it. Traditional DBMS are much faster to answer queries than RAW file parsing. The complexity for simple queries is also sometimes lower than linear, since indexes help to minimise this cost. On the other hand, this solution has the following three main problems.

The first problem of loading the data into a DBMS arises when the data is actually too large to be loaded into the DBMS in a reasonable amount of time. In this case, using RAW file parsing might be faster because it does not need to read, convert, write all converted data back and create auxiliary structures for just then star querying it.

The second problem is that it may be inconvenient to write loading scripts or to define a schema for the data. The work of writing these definitions might be about the same as using RAW file parsing techniques. After all this loading process is done, it is still needed to actually write the querying script, so it might be a much more cumbersome technique.

Third, loading data from a file leads to duplicated data in raw format and in the DBMS format. This requires doubling the required storage, which may not be an alternative if we are dealing with huge data. Versioning and synchronisation problems also derive from this issue: updates to either the raw or loaded data are not automatically propagated to the other side.

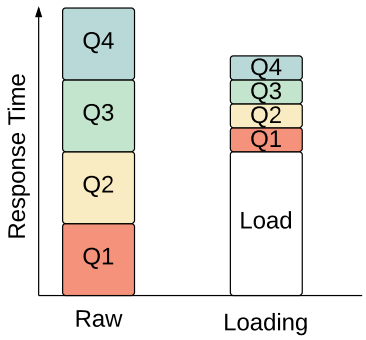

Fig. 3. Differences in response times for queries Q1, Q2, Q3 and Q4, using raw querying and *a-priori* loading.

Imagine four queries Q1, Q2, Q3 and Q4 arriving in an raw parsing and an *a-priori* loading system. As we can see in Figure 3, raw file parsing techniques are better than loading if data is queried only once because it does not have

the data-to-query time of *a-priori* loading. On the other hand, loading the data is better in a scenario where there are multiple, often repeated, queries being processed. This is because contrary to raw parsing techniques, a DBMS provides several optimizations with advanced indexing and caching features. Therefore, it will not read the entire data for each query as in raw parsing.

1.2 Load time versus query time

The presented alternatives are good either for single or several queries because of the different amount of time needed to load or query the data. This introduces the load time versus query time issue. This section tries to explain this issue, and identifies a third, possibly better, alternative.

The two approaches, raw parsing and loading, are opposite solutions to the same problem, and can be classified as seen in Figure 4. The DBMS solution has a high loading time and low query time, because it requires a long time to load the data, but afterwards, all indexing and optimizations of the existing DBMS provide high speed queries. On the other hand, raw parsing does not have any loading time because the query is processed over the original data, but it incurs high query time because they usually do not have indexing features to boost query speed.

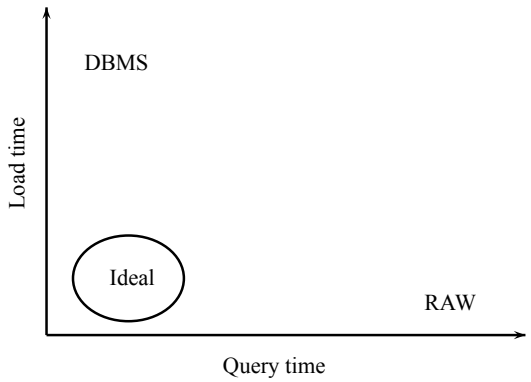

Fig. 4. Load versus query time for DBMS and RAW query approach.

As we can see in Figure 4, there is a desired solution, that provides both low query and load times. This is where current efforts are aiming for. The alternative that might bring this ideal solution relies on hybrid solutions that try to use advantages of both worlds. In the following, hybrid solutions will be discussed.

1.3 Hybrid querying techniques

Hybrid solution might be a better alternative to query raw files. These hybrid solutions share characteristics of traditional DBMS and raw parsing techniques.

The huge advantage of DBMS is their auxiliary indexes that helps to dim the query time. Hybrid techniques will try to use this advantage, creating their auxiliary structures that does not exist in raw parsing techniques. As we will see in the following sections, these will normally be adaptive and incremental to provide better performance according to the workload changes.

In the following sections, techniques that try to approach the ideal performance shown in Figure 4 will be presented. The rest of this work is organized as follows: Section 2 introduces adaptive indexing techniques; Section 3 presents HadoopDB, a distributed DBMS using MapReduce; Section 4 explains an hybrid implementation of a DBMS using raw storage called NoDB.

2 Adaptive indexing

In many deployed RDBMS, the the number of tables and promising indexes exceeds human comprehension and requires automatic index creation. In this context, auto tuning techniques becomes necessary to optimize complex query patterns over huge data sets.

Adaptive indexing focuses on physically adapting the database layout for and by actual queries. The motivation is is that by automatically organizing data the way users request it, it is possible to achieve fast access and self-organized behavior.

Existing techniques for querying raw files like MySQL's CSV engine and Oracle's external tables suffer performance issues because of the lack of indexing features. In this chapter we describe two adaptive indexing techniques: database cracking [6] and adaptive merging [5]. This techniques focus in performance optimization of traditional DBMS, but it may be applied to raw file store since the lack of indexing features is their main pitfall.

2.1 Database Cracking

Database cracking pioneered adaptive indexing. Indexes are created, modified and optimized at fairly low cost as side effect of query execution. The indexing maintenance focus on the key ranges searched in the queries. For example, if most queries focus on lower values of a given column, indexes are never optimized for the higher values.

When a column A is used in a predicate for the first time, a cracker column A_{CRK} is created by copying all data values into it. A_{CRK} is continuously physically reorganized based on queries that uses attribute A, as seen in Figure 5.

Since the cracker column is being continuously split into more pieces as result of the workload, a way to quickly localise a range of interest in A_{CRK} is needed.

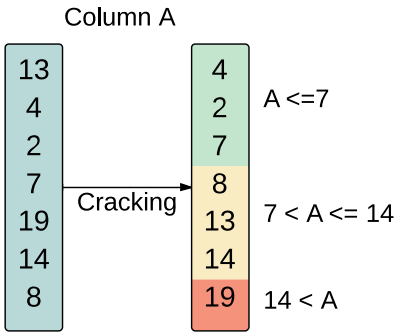

Fig. 5. Cracking a column.

Therefore, each cracker column has a *cracker index* that maintains information on how values are distributed in A_{CRK}. The current database cracking implementation uses an AVL tree where each node holds information for one value, and all values before this node are smaller, and the following ones are grater. The cracker index is refined until sequential searching a partition is faster than binary searching in the AVL tree guiding a search to the appropriate partition.

This information can be used to speed up subsequent queries significantly. Since each time a query arrives the cracker column is split in more slices, it can benefit not only this query, but also similar ones. Therefore, cracking "learns" what a single query can "teach", speeding up multiple queries in the future that request similar, overlapping or even disjoint data of the same attribute.

Cracking was implemented over MonetDB, a column oriented database. Tuple reconstruction in column based DBMS can be done efficiently if the columns are in the same order [3, 9] by avoiding random access. In the cracker column, the original position of the values is compromises due to the physical reordering. Therefore, the column cracker will only be used for value selection, while the original column will continue to be used for tuple reconstruction. Although raw files may not be column-oriented, the copied cracker column still offers benefits in tuple reconstruction, serving just for value selection.

2.2 Adaptive merging

Adaptive merging is an adaptive, incremental and efficient technique for index creation. Index optimization focuses on key ranges used in actual queries, similarly to database cracking, but the resulting index adapts more quickly to new data and to query pattern changes. The main difference from database cracking is that one relies on merging, as in a merge sort, whereas the other relies in partitioning, as in a quick sort.

Like database cracking, adaptive merging requires a flexible underlying storage structure for partially and locally optimized index states. Adaptive merging exploits partitioned B-trees in a different way, aiming to combine efficient merge

sort with adaptive and incremental index optimization. It focus merge steps on key ranges that are appropriate to certain queries, leaving records in all other key ranges in their beginning position.

The B-tree suffers a few improvements. First, an artificial leading key field permits creation and removal of partition identifiers. Second, index creation is divided into run generation and merging. Both can be side effects of query execution or other scans over the data. Third, query execution may optimize such an index by merging the key ranges required to answer actual queries. Forth, non-optimized key ranges are automatically kept in a format that can readily be searched and optimized later if the query pattern changes.

When a column is used in a predicate for the first time, a run generation sorting algorithm is used to append as many partitions as necessary. Each run forms a partition in the new B-tree. Runs are not merged at this time. Their number depends primarily on input size and memory allocation but also on the sort algorithm and any incidental correlation between the sort order in the data source and in the new index.

When a column is used in a predicate for the second time, an appropriate index exists, albeit not yet fully optimized and merged into a single partition. In this situation, a query must find its required records within each partition, typically by probing within the B-tree for the low end of the query range and then scanning to the high end.

These two adaptive indexing techniques are well suited for raw file querying mainly because of their adaptive characteristics. A raw file can be configured to act as the storage layer of a database system, and the indexes may be dynamically created over time. Database cracking is based for column stores, but raw files can still be used independently of its organization since cracking creates a copy of the values in the cracking columns. Same applies to the B-trees on adaptive merging.

In summary, database cracking and adaptive merging offer a promising alternative to traditional index. The great advantage of using adaptive techniques for raw files is that the auxiliary structures can be thrown away if they got outdated with no lost of relevant information. In essence, most indexing technique that benefits traditional DBMS may benefit raw file querying.

3 Hybrid MapReduce

One way to process huge data is to deploy a DBMS on a distributed, shared-nothing architecture. A shared-nothing architecture is a collection of independent machines, each with local disk and local main memory, connected together on a high-speed network connection. DBMS with this kind of architecture tend to scale really well into tens of nodes. However, existing parallel databases start to fail to scale as the number of nodes reaches the hundred or thousand mark.

Existing distributed DBMS fail to scale into higher magnitudes because they implementation often assume that hardware failures are rare, and that the system is deployed a homogeneous nodes. However, with an increasing number of

nodes, failures starts to be common: the probability of one machine failure might be rare, but multiplying it by thousands increases it significantly. The same applies to node homogeneity: it becomes quite difficult to setup and maintain thousands of exactly identical systems.

MapReduce is shared nothing architecture that is well suited to scale to thousand of nodes because it was designed with hardware failures and node heterogeneity in mind. Unfortunately, MapReduce lacks some features of traditional DBMS, like indexing structures. To overcome the limitations of both MapReduce and Traditional distributed DBMS, an hybrid solution called HadoopDB integrates scalability of MapReduce and the useful features of traditional DBMS.

3.1 HadoopDB

HadoopDB [1] is a shared-nothing distributed DBMS. Its goal is to serve as distributed database system that is well suited for the analytical DBMS market and can handle the future demand of data intensive applications. HadoopDB combines the scalability advantages of MapReduce with the performance and efficiency of parallel databases.

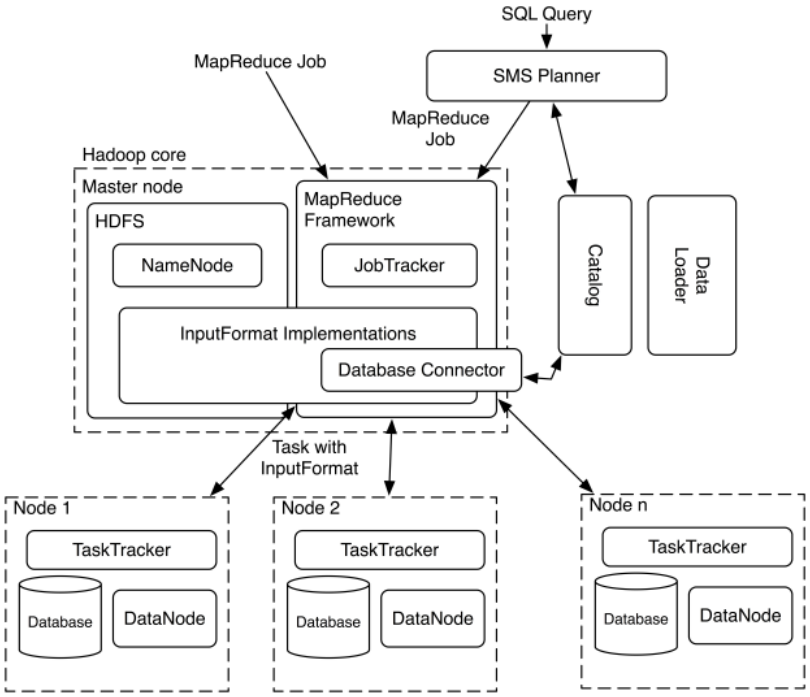

Fig. 6. The Architecture of HadoopDB.

HadoopDB connects multiple nodes running single-node PostgreSQL instances with Hadoop. Hadoop acts as the task coordinator and network communication layer. Queries are expressed in SQL and translated into a special Hadoop job using a modified version of HIVE, a tool that converts HiveQL, a variant of SQL, into MapReduce jobs.

To understand HadoopDB, we need to understand the basics about the Hadoop framework. It consists of two main components. The first is the Hadoop Distributed File System (HDFS). In HDFS, files are broken into blocks, distributed across multiple DataNodes and are managed by a central NameNode. The second component is the master-slave architecture of job handling. There is one master JobTracker that handles scheduling of the MapReduce jobs across the TaskTracker nodes.

HadoopDB extends the Hadoop framework with four new components, the Database Connector, a Catalog, the Data Loader and a SQL to MapReduce to SQL (SMS) Planner, as seen in Figure 6.

The Database Connector extends Hadoop's InputFormat class and is the interface for the single node DBMS. The MapReduce jobs supplies the connector with a SQL query, that is executed and returns results as key-value pairs. At the moment, there are connectors implemented for MySQL and PostgreSQL. This implementations are transparent for the Hadoop framework, which treats the databases data sources as they were data blocks in HDFS.

The Catalog maintains metainformation about the databases such as connection parameters and metadata such as data sets contained in the cluster, replica locations and data partitioning proprieties. The Catalog is stored as a XML file in the HDFS, thus are accessible for the JobTracker and all TaskTrackers.

The Data Loader is responsible for repartitioning data upon loading, breaking data into smaller partitions and loading the single-node databases with data chunks. The partitioning is made with a Global and Local Hasher. The Global Hasher reads raw data stored in HDFS and distributes it among the nodes of the cluster. Local Hasher copies a partition from HDFS into the local file system and partitions it again into smaller sized chunks. This small chunks are then bulk-loaded into the local DBMS instance.

HadoopDB provides a database front-end that allows the processing of SQL queries. The SMS planner extends the Hive query processor. It converts a HiveQL query into an abstract Syntax tree, connects to the Hive catalog to retrieve schemas of tables, creates a DAG of relational operators (the query plan), optimizes it and, finally, the physical plan is converted to one or more Hadoop jobs.

The SMS intercepts the normal Hive flow in two main areas. First, the Catalog is updated with references to the local database tables before any query execution. Second, after the physical plan generation and before the execution of the job, two passes over the physical plan are performed. SMS uses a rule-bases SQL generator to reconstruct SQL from the relational operators, allowing parts of the query processing logic to be pushed into the single database nodes.

Experiments show [1] that HadoopDB is able to approach the performance of parallel database system while achieving similar results on fault tolerance and has the ability to operate in an heterogeneous node environment. The ability to transform any single-node DBMS into a shared nothing parallel database without any code modification is one big advantage of HadoopDB.

HadoopDB provides the possibility to perform distributed queries over several DBMS. One disadvantage of HadoopDB when it comes to raw file querying is that data has to be loaded to the local DBMS using the Data Loader component, thus suffers the same problems as *a-priori* loading solutions. In Section 3.2, a possible solution to optimize loading times of HadoopDB is presented.

3.2 Invisible Loading

Invisible Loading [2] can solve the loading problem of HadoopDB. The concept is to populate a DBMS with the data being queried with Hadoop in run-time. This should be done with minimal human effort, that is, writing a MapReduce job with a fixed parsing API without being forced to specify a complete schema or include explicit database loading operations, and with minimal increase in response time due to loading costs. The idea is to take advantage of the code used for tuple parsing and extraction to invisibly load the parsed tuples into a DBMS. This concept can be compared to use the DBMS as an optimized cache for the raw data.

Invisible loading works as follows: Data is initially stored in the HDFS. Data loading then occurs as side-effect of executing MapReduce jobs over the data. All data loading occurs locally, with no data transfer between nodes, as only local data accessed from the HDFS is loaded on the DBMS. As soon as an input tuple is parsed, the parsed attributes are loaded into the database. To do this, invisible loading jobs are configured with a *Parser* object that implement an *getAttribute(int index)* interface. The map function is then configured to take this parser object as input, instead of the usual key-value pair.

As data gradually migrates from HDFS into the DBMS, the next time the same data needs to be accessed, it can be read faster from the DBMS instead of scanning and parsing data from HDFS. The information of what tuple has already been loaded is kept in the catalog. Hence, as more jobs access the same data set, more of it is migrated to the database system.

To ensure loading performance without visible performance impacts, only a vertical and horizontal partition of the data is loaded per job. The loaded data is also gradually reorganized based on access patterns, using an *Incremental Merge Sort*.

Experiments show [2] that invisible loading provides almost no overhead on MapReduce jobs and optimizes data access for future access without the need of human labor to write loading scripts. However, it still has the problem of data duplication, as in the other loading techniques. Invisible loading does not garbage collect the loaded files because although it stores what tuples has been loaded into the database, it does not have any information if there are more

tuples in a file, thus, there is no automatic way to determine if a particular file has been completely parsed.

MapReduce may be a great solution for distributed computing, but it still have plenty of research area for raw file querying. HadoopDB and Invisible loading does not replaces Hadoop. All systems coexist enabling to choose the appropriate tools for a given dataset and task. The good thing about Hadoop is that it can be combined with existing solutions such as other DBMS to achieve even better results.

4 NoDB

NoDB [4] is a new database system paradigm which do not require data loading while still maintaining the whole feature set of a modern database system. NoDB also provides a series of directives of how to implement auto tuning, caching, and parsing optimizations. PostgresRaw is a NoDB implementation over PostgreSQL that replaces the scan operator with a raw file scanner.

PostgresRaw is a NoDB implementation over PostgreSQL that replaces the scan operator with a raw file scanner. The scan operator has been modified to fetch comma-separated value (CSV) files as tables. When a query referencing an unloaded table arrives, the query plan is modified, changing the scan operator for the raw scan and accesses the respective CSV file. This process can be adapted to turn most modern row-stores into NoDB systems.

Although one of the advantages of using raw files to perform queries is the ability to have schema-less files, PostgresRaw requires the definition of a schema representing the CSV file. This requirement exist because of the nature of PostgreSQL itself, a traditional RDBM. That way it is possible to implement NoDB with a minimum amount of change, maintain the the rest of the Database working as always.

The new scan operator is responsible for parsing and tokenizing of the CSV file. A CSV file consists of a row store. Usually a new line defines the end of a row and a comma the end of a column. Since column sizes are dynamic, each character needs to be processed individually, and with no optimization techniques, this could lead to unpractical performance, requiring the system to parse and tokenize the file every time.

The first possible parsing optimization is selective tokenizing. This technique consists in only tokenizing values that are required for the queries. For example, if a query need the 4th and 8th attribute of a given row, it can be tokenized just to the 8th attribute. Since all data still needs to be read from the file it does not brings any IO benefit. The tokenizing process is a CPU-bound process, and significantly reduces the CPU processing costs.

Parsing is the process of converting the read characters into binary value. Selective parsing can be used to only convert the required attributes into binary data. In addition to this, PostgresRaw also applies delayed parsing, in which the attributes are transformed only when it is know that the tuple qualifies.

The last parsing optimization is the selective tuple formation. It consists of creating tuples that contains only the required attributes.

Selective parsing, selective tokenizing and selective tuple formation does not reduce the IO. Using only this techniques, the entire file would have to be read every time a query arises, even if the query accesses attributes in the same positions. To reduce raw data access, indexing techniques have been used to skip unwanted data and access only the desired attributes, improving significantly data access speed.

PostgresRaw uses an adaptive positional map to keep track of the position of the attributes in a given CSV file. This positional map stores metadata information such as byte offset of the attributes in the CSV file. If a query needs this attribute, the scan operator can skip directly to the attribute position, significantly reducing IO, tokenizing and parsing costs.

The adaptive position map is dynamically populated at run-time. Every time an attribute is read, its position is stored in the index. This index can, therefore grow indefinitely, thus some maintenance is required to allow the index to be practical. When the index exceeds the maximum defined size, a LRU policy acts to trow away old indices and give space to the new ones.

Adaptive indexing such as database cracking and adaptive merging can be useful to manage the positional map. If these techniques are applied correctly to the implementation, they can improve the performance even more.

Updates in the CSV file should be reflected on the positional map. Updates that do not change column size are trivial, but if there is a column size change, or if records are deleted or added in the middle of the file, all the positions after this point of change should be updated. There are to options in this case. The first is to analyse each change case and recalculate the new positions of the attributes. The second is to simply partially or completely drop the map structure. Since the index is an auxiliary structure, it can be removed at any time without loosing critical information and will be regenerated over time.

The positional map offers great performance benefits when data needs to be accessed from the file. Nevertheless, further performance optimizations can be achieved if the data does not need to be fetched from the file at all. To accomplish that, traditional caching techniques can be used.

Other traditional optimization technique used in PostgresRaw is to collect statistics in order to produce more efficient query plans. The new raw scan operator provides insightful statistics that can be used by the traditional DBMS. This technique takes again advantage of replacing just the required part of the DBMS, and to try to keep using all the already working optimizations.

PostgresRaw offers a great hybrid of DBMS and raw files, and experiments show [4] that it is capable to provide competitive performance with traditional database systems. It is able to eliminate loading times, using the raw file as storage, but can also provide fast queries thanks to all the auxiliary structures.

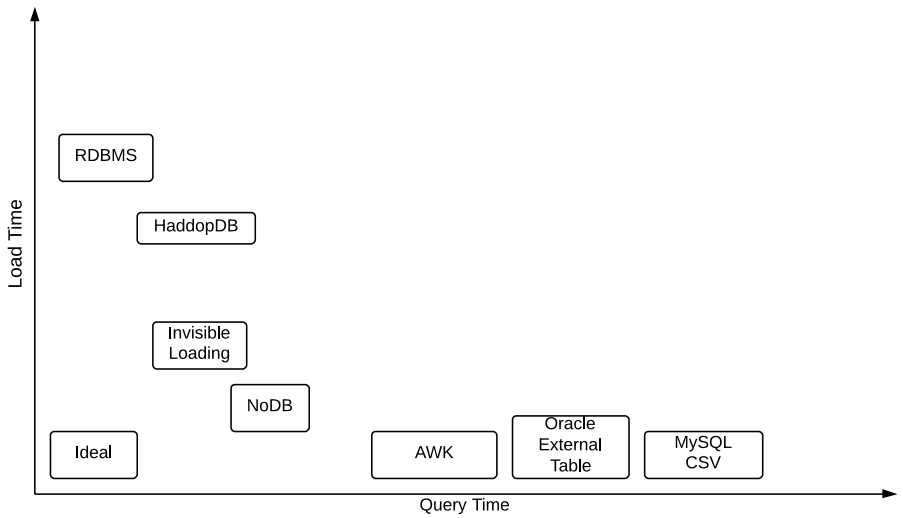

Fig. 7. Load versus query time of several querying techniques.

5 Conclusions and Future Work

In this paper we presented several scenarios where raw files might be useful: When there is too much data to be processed like Big Data and scientific research, to improve interoperability or avoid repetitive human labour such as writing loading scripts and schemas.

The current mature solutions to query raw data suffer from either high load times or high query times. We identified that high query times occurs because of the lack of indexing structures of the raw data, and high load times occurs when trying to load all data before query execution. Hybrid solutions usually tries to bring indexing features and *in-situ* processing over raw data.

Hybrid techniques creates the opportunity to remove the main bottleneck of precessing very large data, the load time. With hybrid techniques, it is possible to exploit databases technology without the need of loading data by enabling *in-situ* querying, and removing high query times by using indexing features.

Adaptive indexing can be an important pillar of raw data querying, since lack of indexing is one of the main pitfalls of raw file querying. Cracking was the pioneer implementation of adaptive indexing and adaptive merging is an evolution of this implementation. Both show incremental improvements in query times for repetitive query patterns.

HadoopDB offers a distributed DBMS over a shared-nothing architecture provided by the Hadoop framework. It offers a reliable and scalable solution because it uses MapReduce as the communication layers. HadoopDB still needs to load the files in order to benefit from his distribute architecture. Invisible loading provides a way to dim loading times by doing incremental load of the

data, gradually copying the data from the Hadoop Distributed File system into single node DBMS instances.

NoDB provides a series of directives on how to implement a DBMS with a raw file storage system. PostgresRaw is a NoDB implementation over PostgreSQL that replaces the scan operator with a CSV file reader. Several optimizations are made to reduce query time, such as selective parsing, tokenizing and tuple formation, advanced indexing for the raw file, caching and statistics over the raw file. All optimizations of PostgresRaw makes it able to provide competitive performance with traditional database systems.

In figure 7, discussed techniques are approximately organised according to their load and query times. As we can see, raw file querying such as AWK, Oracle external table and MySQL CSV engine have no loading times, but query times are high. In contrast, traditional DBMS systems have very low query time, but high loading times for *a-priori* loading. Hybrid techniques tries to overcome this limitation. HadoopDB has a lower query time than the raw file parsing techniques, but high load times. Invisible loading tries to reduce the loading time for HadoopDB, and NoDB offers a good balance betweem query and load times. We can see that this solutions are converging to the ideal introduced in figure 4, but in essence, there are still open issues and research ground for querying raw files.

References

1. Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.
2. Azza Abouzied, Daniel J. Abadi, and Avi Silberschatz. Invisible loading: access-driven data transfer from raw files into database systems. *Proceedings of the 16th International Conference on Extending Database Technology*, pages 1–10, 2013.
3. Peter A Boncz and Martin L Kersten. MIL Primitives For Querying A Fragmented World, 1999.
4. Renata Borovica, Stratos Idreos, and Anastasia Ailamaki. NoDB : Efficient Query Execution on Raw Data Files Categories and Subject Descriptors. pages 241–252.
5. Goetz Graefe and Harumi Kuno. Adaptive indexing for relational keys. *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 69–74, 2010.
6. Felix Halim, S Idreos, P Karras, and RHC Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *Proceedings of the VLDB Endowment (PVLDB)*, 5(6):502–513, 2012.
7. Tony Hey, Stewart Tansley, and Kristin Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, Redmond, Washington, 2009.
8. Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. Here are my data files. here are my queries. where are my results. *Proceedings of 5th Biennial Conference on Innovative Data Systems Research*, pages 57–68, 2011.
9. Stefan Manegold, Peter Boncz, Niels Nes, and Martin Kersten. Cache-Conscious Radix-Decluster Projections. In *In Proc. VLDB*, 2004.

10. Christopher Olston, Benjamin Reed, Ravi Kumar, and Andrew Tomkins. Pig Latin : A Not-So-Foreign Language for Data Processing. pages 1099–1110.
11. Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.