Invisible Loading: Access-Driven Data Transfer from Raw Files into Database Systems

Azza Abouzied
Yale University
azza@cs.yale.edu

Daniel J. Abadi
Yale University
dna@cs.yale.edu

Avi Silberschatz
Yale University
avi@cs.yale.edu

ABSTRACT

Commercial analytical database systems suffer from a high "time-to-first-analysis": before data can be processed, it must be modeled and schematized (a human effort), transferred into the database's storage layer, and optionally clustered and indexed (a computational effort). For many types of structured data, this upfront effort is unjustifiable, so the data are processed directly over the file system using the Hadoop framework, despite the cumulative performance benefits of processing this data in an analytical database system. In this paper we describe a system that achieves the immediate gratification of running MapReduce jobs directly over a file system, while still making progress towards the long-term performance benefits of database systems. The basic idea is to piggyback on MapReduce jobs, leverage their parsing and tuple extraction operations to incrementally load and organize tuples into a database system, while simultaneously processing the file system data. We call this scheme *Invisible Loading*, as we load fractions of data at a time at almost no marginal cost in query latency, but still allow future queries to run much faster.

1. INTRODUCTION

There are many types of data that, despite being structured enough to fit into a relational model, are stored in flat files on a file system instead of in a database system [11]. Examples include logs (especially network event logs), machine output from scientific experiments, simulation data, sensor data, and online click-streams. Much of this data is append-only and usually analyzed many times over the course of their lifetime. Many of the features available in database systems, such as ACID guarantees, are often not needed. Moreover, database systems require that a schema be clearly defined, and data be loaded into the system before it can be used, a time and effort overhead often deemed unnecessary for this type of data.

The types of data sets listed above can often be extremely large (terabytes to petabytes in size), and therefore distributed file systems are increasingly being used for storing them and even serving as an analytical platform. Perhaps the most well-known of these systems is Hadoop, which bundles an open source version of Google's distributed file system called HDFS with an implementation of a MapReduce framework for data analysis. Hadoop (and the rapidly growing ecosystem around it) is becoming increasingly popular as a platform for data analysis.

Hadoop is extremely scalable and has a low "time-to-first analysis": as soon as data are produced, they are available for analysis via MapReduce jobs. This is in contrast with database systems that require data to be loaded before SQL queries can be run. Hadoop trades cumulative long-term performance benefits for quick initial analysis: recent work comparing the performance of Hadoop with database systems demonstrates that once data have been loaded, database systems take advantage of their optimized data layout to significantly outperform Hadoop[15].

Data preparation for database systems involves a non-trivial human cost (data modeling and schematizing) that is quite different from the tunable computational costs of copying, clustering and indexing[11]. If a user is not intimately familiar with the data and understands the meaning of only a few fields, he/she is unlikely to take the responsibility of generating a schema for the data. For example, a new member of a research group that inherits a simulation program written by a PhD student who has since graduated is unlikely to understand the program well enough to generate a schema for the program's output. Similar issues exist for a scientist that wants to analyze the output of experimental data produced by a machine whose manufacturer's documentation is unavailable, or a systems administrator who understands the meaning of only the first few fields. These users prefer to work in schema-free environments, writing scripts to process only the fields they understand. Furthermore, going the through the documentation of the database system to figure out the right commands to load the data from the file system to the database system is an additional annoyance that users prefer to avoid.

Our goal in this paper is to describe *Invisible Loading*: a scheme that achieves the low time-to-first analysis of MapReduce jobs over a distributed file system while still yielding the long-term performance benefits of database systems:

1. We provide a mechanism for users to separate parsing code from data-processing code in MapReduce jobs. Users are not required to parse any additional attributes beyond what they already are familiar with and know how to parse (see Section 2). (If the data is already stored using Avro, ProtocolBuffers, or HCatalog, this step can be avoided).

2. We reduce the loading overhead by only copying some vertical and horizontal partitions of the data into the underlying column-store database system (in this paper we use MonetDB [13]). We introduce two new points in the 'upfront loading overhead cost vs. better cumulative performance'

EDBT/ICDT '13, March 18 - 22 2013, Genoa, Italy

tradeoff space, and compare them experimentally with one another and with traditional implementations that sit at the extreme edges of this tradeoff (see Section 3).

3. We introduce an *Incremental Merge Sort* technique to incrementally reorganize data based on selection predicates used to filter data (see Section 2.2).

4. We manage the loading, reorganization, and querying of different columns that are at different stages of the loading or reorganization phases. (see Section 2.3).

We also provide a library of operators that are capable of processing data no matter where they might be located (in the distributed file system or in the database system). User jobs written using these libraries achieve the most performance benefits (see Section 2.1.1).

We focus on the problem of loading data from a *distributed file system* to a *shared-nothing parallel database system*. We require that the database system and the distributed file system are located on the same set of nodes in the cluster. We can therefore load data from each node of the distributed file system to the corresponding node of the parallel database system without shipping any data over the network. This allows us to avoid the problem of global data sorting: we focus only on how to cluster data locally within each node of the parallel database system. Therefore, the techniques presented in this paper are also applicable to loading data from a single machine file system to a database system sitting on the same machine.

Before we discuss the architecture and implementation details of Invisible Loading, we provide a primer on the components of a Hadoop job and column-stores in the next section.

1.1 Background

The Anatomy of a Hadoop Job. The main components of a Hadoop job are its *map* and *reduce* functions. A *map* function processes **one** key-value pair at a time to produce zero or more key-value pairs. On each Hadoop node, a Map task executes the *map* function over an individual *split* of the data set. If a *reduce* function is specified, the key-value pairs produced by the Map tasks are sorted and shuffled by key across one or more Reduce tasks. The user could also optionally specify *configure* and *close* procedures. A Map task executes the *configure* procedure before the *map* function. After the *map* function consumes its entire data split, the *close* procedure is executed. The user also specifies the data set to process, which could be one or more files that are managed by HDFS and the *InputFormat* to use. The InputFormat determines how a data set is logically split and how to read in its contents. Many InputFormats do not provide parsing functionality: in these cases, parsing is intermixed with data processing in *map* functions.

Column-store Database Systems. Column-stores map relational database tables to disk column-by-column instead of row-by-row: this enables each column to be accessed independently without having to waste time reading other irrelevant columns. The savings in I/O costs often yield significant performance improvements for analytical workloads that scan only a subset of database columns per query. However, for point lookups and updates, a column-oriented data layout is not optimal. Although column-stores store each column separately, they must still retain the ability to reconstruct tuples by stitching together values from different columns. The most common technique is to make sure that the i^{th} value in each column comes from the i^{th} tuple in the relational table. Tuple reconstruction can then quickly occur by simply doing a linear merge of the columns. Another technique is to maintain a tuple identifier (OID) with each column value, and use this identifier to match up values across columns.

2. INVISIBLE LOADING

Our goal is to move data from a file system to a database system, with minimal human intervention (writing MapReduce jobs using a fixed parsing API) and without any human detection (no visible increase in response time due to loading costs). Our idea is to piggyback on special MapReduce jobs written by users, leveraging the code used for tuple parsing and extraction to invisibly load the parsed data tuples into the database system. Our specific goals are: (i) The user should not be forced to specify a complete schema, nor be forced to include explicit database loading operations in MapReduce jobs. (ii) The user should not notice the additional performance overhead of loading work that is piggy-backed on top of the regular analysis.

Invisible Loading works as follows:

1) Data are initially stored in the Hadoop Distributed File System (HDFS). These data are immediately available for analysis using Invisible Loading (IL) Hadoop jobs. Database loading occurs as a side-effect of executing MapReduce jobs over the data in HDFS. We leverage the runtime scanning and parsing performed by a job to simultaneously load the parsed data into a database system. Section 2.1.1 discusses implementation details that improve loading performance and ensure consistency. Each time a node accesses local HDFS data, we load the local partition of the parallel database system with these data. Therefore, all loading occurs **locally** with no data transfer between nodes.

2) We ensure users do not notice a loading cost overhead by only loading a vertical and horizontal partition of the data per job (Section 2.1.2). We also reorganize data gradually based on data access patterns (Section 2.2).

3) As data gradually migrate from HDFS into the database systems, jobs are redirected to access pre-parsed horizontal data partitions from the database for their input instead of scanning and parsing data from HDFS. Hence, as more jobs access the same data set, more of it is migrated to the database system, resulting in performance improvements due to the more efficient data access provided by the database system (Section 2.3).

We now describe in detail certain aspects of our system implementation:

2.1 Implementation Details

The core of the Invisible Loading system is an abstract, polymorphic Hadoop job, *InvisibleLoadJobBase*, that hides the process of data loading from the user. It is abstract, since it requires users to implement the parsing and processing functions of a *map* function, and it must be configured like a typical Hadoop job. It is polymorphic, since it dynamically self-configures to modify its behavior as data migrate from the file system to the database systems. We will refer to a concrete extension of the InvisibleLoadJobBase as an IL job.

2.1.1 Leveraging Parsing Code

Our first objective is to leverage the parsing code that exists within the *map* function for database loading. In particular, we would like to inject a load statement in between the parsing and processing phases of the *map* function: as soon as an input tuple is parsed, we load the parsed attributes of the tuple into the database and then continue processing it. There are two approaches to achieving this objective. One approach is to perform complex static code analysis, such as used in HadoopToSQL [12] or Manimal [3], to differentiate the parsing from the processing code segments of a *map* function and then rewrite the function to parse,

load, process. Analyzing code at a logical level, however, is non-trivial and error-prone: parsing can be mistook for processing of a single-attribute tuple.

We opt for a simpler approach: we impose a parse-process structure on IL jobs. The user configures the IL job with a *Parser* object that reads in an input tuple, extracts the attributes from the tuple relevant for the analysis, and returns these attributes through a simple *getAttribute(int index)* interface. The user then writes the *map* function to take in as input the parser object instead of the usual key-value pair. The IL job manages the flow of input tuples through the parser object into the load process and finally into the *map* function defined by the user.

Tables in the database are identified by the source data set and the *Parser* implementation used to extract the data. This results in the following flexible schema for any data set with n attributes:

```
Table name: <file_name>_<parser_name>;
Schema: (1 <type>, 2 <type>, ...,
         n <type>);
```

Since different parsers could extract different tuples from the same data set, we store different tables for different parser - data set combinations. Section 2.1.2 discusses this in detail.

The underlying *column-store* database system generates a hidden address column to maintain a mapping from the loaded data to the HDFS file-splits. The initial loading sequence of the file-splits fixes the address range associated with each split: If a file with two splits F_1, F_2 has x tuples in each split and the initial loading sequence was F_2, F_1, then the address range $[0 - x)$ is associated with split F_2 and the address range $[x - 2x)$ is associated with split F_1. The address ranges associated with each split are stored in a catalog. The address column enables the alignment of partially loaded columns with other clustered columns (see Section 2.3).

The catalog also maintains a mapping between a data set and one or more tables that contain data loaded from the data set but were extracted using different Parsers, and keeps track of loading progress.

The *configure* function of an IL job first checks to determine if an entry exists for a particular data set - parser combination. If not, it issues a SQL CREATE TABLE command. If an entry exists, it determines which HDFS file splits and attributes have been loaded into the database system. If the required data is already loaded, the IL job self-configures to read its input from the database system. Since data in the database system are pre-parsed, the IL job simply replaces the *Parser* object with a Dummy Parser. Otherwise, the IL job parses and loads the data and then applies the *map* function. There are two implementations of the injected load operation: a *direct load* and a *delayed load*.

In a direct load, we immediately load the parsed attributes of every input tuple as soon as it is scanned and parsed. We only utilize a direct load if the underlying database system enables efficient streaming inserts. After the *map* function consumes all inputs from a split, the *close* procedure updates the catalog with information about the HDFS splits and the attributes loaded, as well as the *Parser* implementation used to extract the attributes and then commits the inserts. This ensures consistency through an atomic load: if a map task fails, then no tuples are committed into the database and on a re-run of the map task, loading will not result in duplicates.

In a delayed load, we simply write the parsed attributes to a temporary memory buffer (spilling to disk if necessary) and in the *close* procedure we execute a SQL 'COPY' command to append the file from the memory buffer into the database table. For the MonetDB system we found that the delayed load was more efficient than the direct load.

Imposing a parse-process structure is common in several MapReduce environments: users explicitly define the parsing process in PigLatin[14] using "LOAD ... USING <parser> AS ..." and in SCOPE[4] using "EXTRACT ... FROM ... USING <parser>".

Prior work on systems that split execution across Hadoop and database systems [1] have found that in order to achieve the full performance potential of using database systems as the storage layer instead of HDFS, the system must push as much as possible of the processing into the database (instead of simply using the database for data extraction). We provide a basic library of operators, such as *filter*, *aggregate*, ... etc, that behave differently depending on the data source. Several MapReduce programming environments, like PigLatin [14], provide similar libraries of operators. Such libraries generally facilitate the user's programming task.

Even if a user chooses not to use these libraries, the efficient data scans provided by column-store database systems provide a performance advantage over vanilla Hadoop scans. Scanning two columns out of five from a 2GB data set using a database system takes a 170 seconds, whereas it takes 300 seconds in Hadoop.

2.1.2 Incrementally Loading Attributes

Our second objective is to incrementally load attributes instead of loading all attributes extracted by the *Parser*. By only loading the attributes that are actually processed, we reduce the overall overhead of loading per job. In addition, we do not waste effort loading attributes that are never processed by any Hadoop job. Furthermore, users might prefer not to write a complete parser for a data set if they are not intimately familiar with how data were generated. Therefore, loading can occur despite incomplete or incorrect parsing code.

Since the catalog contains information on the loaded attributes, the IL job utilizes this information to determine which attributes if any, need to be loaded into the database. If new attributes need to be loaded, the *configure* procedure of each Map task issues an ALTER TABLE command to modify the database schema.

We illustrate the process with an example: two IL jobs are executed in sequence, the first job processes attributes a, b, c and the second job processes b, c, d. When the first job terminates, the database is loaded with a horizontal partition of the attributes a, b, c. The catalog reflects the current database state. When the second job executes, it examines the catalog and intersects the set of partially loaded attributes with the set of attributes it processes and determines that a horizontal partition of attribute d needs to be loaded. It, then, self-configures its load operations to include attribute d. The *configure* procedure issues an "ALTER TABLE ... ADD COLUMN (d, ...)".

The size of the horizontal partition loaded per IL job is a system-configured parameter and is specified as a fraction of the HDFS file. If the system-configured parameter is $1/8$ and an HDFS file has 32 splits, then each job will load an unloaded horizontal partition of a subset of attributes from four file splits.

To efficiently add a column to an existing table, we use column-stores. A traditional row-oriented database physically stores a tuple's attribute-values together. Therefore, altering the schema typically requires a complete re-write of the table in order to make room for the newly loaded (not null) attributes. However, a column-store does not need to physically restructure the table, since each column is stored separately.

2.2 Incremental Data Reorganization

So far, we have only discussed data loading in terms of data copy from the file system to the database system. We now explain how we incrementally reorganize data based on selection predicates

used in the filter operator we provide for users.

In general, in order to optimize the performance of a traditional database, a (highly paid and skilled) database administrator determines which indices are necessary given the current query workload. Index selection is a hard database design problem: wrong indexes result in poor query execution plans and high data update costs. Moreover, ad-hoc queries complicate the administrator's task. To address this problem, self-tuning databases monitor query workloads and query execution plans and create or remove indices in response [5]. Traditional indices, however, offer an all-or-nothing service: until data are completely indexed, no data access benefits exist, and while data are being indexed, the overhead of indexing either interferes with the query execution or brings querying to a standstill (if tables need to be locked). In addition, complete index creation can take days to complete depending on the size of the data set.

Our technique, *Incremental Merge Sort*, is based on the basic two-way external merge sort algorithm[1].

The basic two-way external merge sort works as follows:

1) First, a column of n tuples is broken down into k slices, such that each slice fits in memory. For simplicity, assume k is a power of 2, $k = 2^a$.

2) Each slice is then sorted in-memory using any efficient, cache conscious, in-memory sorting algorithm such as quicksort or radix-sort

3) Two slices are merged at a time to create a larger sorted slice. The entire column is sorted after $k - 1$ merge operations. In the first phase, $k/2^1$ merge operations perform $2 * n/k$ tuple comparisons each. In the second phase, $k/2^2$ merge operations perform $2^2 * n/k$ comparisons and so on. The last merge operation performs n comparisons: $2^a * n/k = 2^a * n/2^a = n$.

The standard two-way external merge sort performs for every merge operation in a given phase, twice the amount of work it performed in each merge operation of the previous phase. Hence, the amount of effort, measured as query response time, exponentially grows until the data are completely sorted. This exponential growth behavior defeats the key feature of any incremental strategy: *perform equal if not less effort for any query in comparison to the previous query*. Therefore, to maintain a monotonically decreasing query response time, each merge operation has to perform a bounded number of tuple comparisons. Our variant operates like external merge-sort with an added split step:

1) Partition a column of n tuples into k sorted slices. In our initial load step, we maintain a one-to-one relationship between logical file splits and sorted slices. After sorting each slice, we calculate the range of the data. i.e. the smallest and largest value. From this range, we calculate a *split-bit* for the first phase of incremental merge sort. This is equivalent to the highest power of two that is less than the largest value. Therefore, if our data range is [0,15], the split-bit is 8_{10} or 1000_2.

2) The algorithm then goes through $\log k$ phases of $k/2$ merge and split operations that process on average $2 * n/k$ tuples. We perform the merge operation as usual except we split our results into two new slices based on whether the logical bit-wise AND of the tuple and the split-bit is 0 or 1. This step is similar to the partitioning step in radix-sort. Once a phase is complete, the split-bit is right-shifted (or divided by two), for the next phase. This ensures that the two newly created slices represent disjoint ranges where one key range is larger than the other. At the end of the first phase, exactly half the slices contain key ranges larger than the split-bit — this represents a *major* partition in the data: slices from one partition need not be merged with slices of the other partition as they do not overlap. At phase p, where $p \in [1, \log k]$, a slice i, is merged with slice $i + p$, if i has not already been merged.

[1]We describe all algorithms in terms of sorting an integer column, even though the approaches extend to any data type.

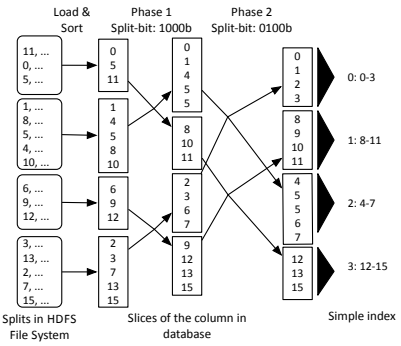

Figure 1: Incremental Merge Sort on a column of four slices.

3) After $k/2 * \log k$ steps, the data are completely ordered with each slice containing a contiguous data range. A simple index holding only k entries describes the distribution of key ranges in each slice.

Figure 1 illustrates the operation of incremental merge sort over four slices of slightly skewed data in the range [0, 15].

The selection predicate determines which column is used to organize the entire data set. At data loading, we determine this column based on the filtering predicate used in the IL job. For example, if the first job processes tuples with attributes a, b, c, d that pass a selection predicate on a, then we order the data set on attribute a. If another job processes tuples with attributes b, d that pass a predicate on b, we create a physical copy of columns b, d and we incrementally order these columns on b. This approach is similar to cracking [9, 10], which creates a new cracker column every time an attribute is used as the selection attribute. See Section 2.3 for more details. Our strategy is not as adaptive as cracking since it is not query driven: all tuples are treated with equal importance even if a certain key range is more heavily queried. There is a straightforward extension to this basic algorithm that allows the adaptive merge and split of slices that fall within a heavily queried key range.

If data are skewed, certain slices can be much larger than others. This means more effort is spent merging and splitting these slices. To ensure a bounded amount of re-organization work per query, we spread each merge-split operation over multiple queries by bounding the number of tuples processed by each query to at most $2 * n/k$ and preserving a cursor that points to the next tuple that needs to be processed in a partially-processed slice.

The final output after all incremental reorganization has taken place is a set of sorted runs and an index over them that can be used to create a completely sorted scan of the data. If the column is memory resident, the process stops here (instead of shuffling around the slices to ensure consecutive slices are contiguous), since slices are sufficiently large, and the overhead of a random in-memory jump to the next slice (when scanning a column in sorted order) is small enough that it does not warrant the complete copy of a column to ensure consecutive slices are contiguous. However, if the column is not memory resident, the overhead of a random disk seek when reading slices from disk is much larger. Therefore, the slices are reorganized step-by-step (using a straightforward incremental process) to ensure a contiguous sorted order on storage.

Incremental Merge Sort easily integrates compression. Since slices are sorted as soon as they are loaded, run-length encoding (RLE) could be applied immediately to the sorted column. This not only reduces the size of the slice but also the number of subsequent comparisons that occur while merging data. Differential compression, dictionary encoding and similar techniques that represent data with order preserving symbols of fewer bits could be applied to reduce the memory size of each slice. The choice of which compression scheme to use depends on whether the system is I/O or CPU limited. More aggressive compression schemes will fit more data per slice benefiting an I/O limited system but at the expense of higher CPU costs, especially if decompression is necessary for processing. In section 3.3, we empirically evaluate the benefit of integrating a lightweight compression scheme with data reorganization. In addition to data compression benefits, Incremental Merge Sort also supports reorganizing variable-width data as the merge operation reads data from the slices in a single pass in sequential order and sequentially writes data to new slices in a single pass.

2.3 Integrating Invisible Loading with Incremental Reorganization

The frequency of access of a particular attribute by different users determines how much of it is loaded into the database system. Filtering operations on a particular attribute will cause the database system to sort the attribute's column. Therefore, as different users submit jobs that access different subsets of the data, the set of loaded columns could diverge from each other in terms of completeness and order.

We rely on two basic tools, address columns and tuple-identifiers (OIDs) to manage the querying of columns at different loading and reorganization stages. We use address columns to track the movement of tuples, due to sorting, away from their original insertion positions. We use tuple-identifiers (OIDs) to determine how much of a column has been loaded and to align the column with other columns that have different sorting orders.

The following simple rules deal with the different loading and reorganization states different columns could exist in:

1) If a set of columns are completely loaded and sorted with the same order, then they are all positionally aligned with each other and a simple linear merge suffices when reconstructing tuples from these columns.

2) Columns that are partially loaded have their OIDs in insertion order. To reconstruct tuples from completely loaded (and perhaps reorganized) columns and partially loaded columns, a join is performed between the address column of the index column and the OIDs of the partially loaded columns.

3) If a column needs to have a different sorting order, then a copy of that column is created (and other dependent columns). An address column is generated to track the movement of tuples from their original insertion orders to their new sorting orders.

We illustrate how we integrate invisible loading with incremental reorganization through the application of these rules using case-by-case examples of different combinations of queries from three different users named: X, Y, Z.

Consider the data set with four attributes a, b, c, d. User X is interested in attributes $\{\bar{a}, b\}$, where $\bar{a}$ denotes a selection predicate on a. User Y is interested in attributes $\{\bar{a}, c\}$. User Z is interested in attributes $\{\bar{b}, d\}$. Assume the file has only four splits per node and the horizontal-partitioning fraction is $1/4$, so at most one split is loaded per job per node. The number of slices k is equal to the number of splits.

Case 0: XXXX-YYYY. For each X query a horizontal partition of attributes $\{a, b\}$ is loaded and immediately sorted by attribute a. After four of user X's queries, attributes $\{a, b\}$ are completely loaded. The hidden address column tracks the movement of tuples due to the (i) individual sorting of slices by attribute a that occurred after loading each slice and (ii) the merge-split operations that were triggered by the third query as two slices were already loaded by then. Since b is positionally aligned with a, the tuple identifier (OID) values of a, b are not materialized.

Query Y starts the loading of attribute c. The first query Y is processed entirely from the file system with a single partition of c loaded into the database.

The second Y query will execute the following relational expression in the database system ($f(a)$ is a predicate on a):

$$\pi_{a,c}\left(\sigma_{f(a)(a,addr_a)} \bowtie_{(oid_c,c)}\right) \quad (1)$$

The OIDs associated with column c fall within the address range associated with the first loaded partition. The remaining three splits are processed entirely over the file system and a second partition is loaded. All completely loaded columns that depend on a, namely b, are kept aligned with a. Therefore we make use of the two tuple re-construction techniques: positional alignment when retrieving tuples from a and b and tuple-identifier (OID) matching when retrieving tuples from a and the partially loaded column c.

After four of user Y's queries, column c is completely loaded. The database system then positionally aligns column c with column a and drops its materialized OID values.

We evaluate the effect on query response time of loading an additional attribute to a table in Section 3.2.2. We compare this strategy with the approach of loading an entire column in one job and immediately aligning it with the other completely loaded columns.

Case 1: XX-YYYY-XX. The first two X queries cause the system to behave as in case 0. The following two Y queries will load only attribute c from the first two file-splits. Column c has materialized OID values and is not positionally aligned with column a. The relational expression (1) (above) is used to query columns a, c.

After the third Y query, another partition of attributes a, c is loaded. The newly added slice of column a is immediately sorted and the first two loaded slices of a are merge-split. Since column b is no longer aligned with a, its OID values are materialized from the address column. After the fourth Y query, a and c are completely loaded. Column c is then positionally aligned with a, and its OID values are dropped.

Case 2: {Case 0 | Case 1} - ZZZZ. The first query Z loads a partition of column d into the database system. The second query Z selects tuples $\{b, d\}$ filtered by b from the loaded database partition of $\{b, d\}$ using the following relational expression: $\pi_{b,d}\left(\sigma_{f(b)(b,addr_a)} \bowtie_{(oid_d,d)}\right)$.

The selection on column b initiates the incremental reorganization of column b. A copy of b and the address column is created - $b', addr_b$. Column b' is then sliced, with each slice individually sorted and so on as specified by incremental merge sort. Column $addr_b$ keeps track of the movement of values within column b'. After the four Z queries, column d is completely loaded and is positionally aligned with b'. Any future Z queries will be satisfied using columns b', d.

Case 3: XX-ZZZZ-XX. After the fourth Z query, the database system has a partially loaded column a with an associated address column $addr_a$. Columns b', d are completely loaded with address column $addr_b$. The following X queries load the remaining partitions of attribute a. The equivalent partitions of column b are copied from within the database system using the following ex-

pression: $\pi_{b'}\left(_{addr_a}\bowtie_{(addr_b,b')}\right)$. Columns a, b are, therefore, kept positionally aligned.

3. EXPERIMENTS

We ran several experiments in order to carefully examine the tradeoffs and performance of invisible loading. These experiments use a complete load and sort of the entire data set into the database system ("SQL Pre-load") and MapReduce jobs over the file system ("MapReduce") as comparison points against which to compare invisible loading. It is expected that SQL Pre-load will incur a significant upfront performance overhead, but will have long term performance benefits, while MapReduce will have zero upfront performance overhead, but very poor cumulative performance over the course of many queries. Therefore, we compare various versions of invisible loading that serve as a compromise between these extremes (the different loading and data reorganization strategies that we will experiment with are outlined in table 1). These versions of invisible loading vary the fraction of data loaded per MapReduce job along two dimensions: vertically, (i.e. only load a subset of the parsed attributes) and horizontally (i.e. only load horizontal partitions of the data set).

We evaluate both the short- and long- term benefits of each approach by measuring both query response time and cumulative performance. The experiments consist of a sequence of MapReduce jobs that perform similar analysis repeatedly over the same data set. We chose this simplistic experimental workload because the particular type of analysis does not affect our strategies, all that matters is the data access patterns. We vary our data access patterns through changing the column projections and tuple selection predicates and present these extended results in Section 3.2.2.

We then evaluate with much finer detail our incremental merge sort data reorganization strategy. We compare it to pre-sorting and cracking, MonetDB's data reorganization technique [9]. Since all schemes have already copied the data into the database system before data reorganization begins, this set of experiments is run entirely within the database system so that the differences between the different approaches can be entirely attributed to the incremental reorganization algorithms.

3.1 Experimental Setup

Hardware: All experiments were run on a quad core machine with 12GB of main memory, a single 250GB disk partition and a single 200MB RAM disk partition. A single machine is sufficient for the following reasons: all our loading and data re-organization operations occur locally during the map phase. All strategies that we evaluate require that the parallel database be configured such there there is a one-to-one correspondence of distributed file system nodes to parallel database nodes (these nodes are collocated on the same physical machine). All strategies compared in the experiments are "embarrassingly parallel": there is nothing extra to be learned from running more than one of these completely independent tasks (there is no global redistribution of data whatsoever). Even loading catalogs are maintained locally — there is no non-linear cost associated with updating a centralized catalog.

Software: Hadoop (0.19.1) was used to evaluate the performance of MapReduce executing over a distributed file system (HDFS). MonetDB 5[2], was used as the column-oriented database system into which data were loaded from HDFS. All IL jobs utilize a library of basic functions: a *TabParser*, and a *Filter* function. The maximum number of concurrent Map or Reduce tasks was configured to one. This restriction was imposed to enable accurate profiling of the systems involved without the overhead of processes competing for I/O or memory resources. All systems were injected with appropriate profiling statements to keep track of the precise time spent in loading, sorting, merging, scanning or selecting data.

[2]MonetDB: 20-Jan-2010 developer's build

	Strategy	Description
1	SQL Pre-load	Pre-load the entire dataset into the database using SQL's 'COPY INTO' command. Data are sorted after loading using 'ORDER BY'.
2	Incremental Reorganize (all)	Load the entire dataset into the database system upon its first access, but unlike Pre-load above, do not immediately sort the data. Instead, data are incrementally reorganized as more queries access the data.
3	Incremental Reorganize (subset)	Same as Incremental Reorganize (all), except that only those attributes that are accessed by the current MapReduce job are loaded.
4	Invisible Loading (all)	The invisible loading algorithm described in Section 2, except that all attributes are loaded into the database (instead of the subset accessed by a particular MapReduce job).
5	Invisible Loading (subset)	The complete invisible loading algorithm described in Section 2.
6	MapReduce	Process the data entirely in Hadoop without database loading or reorganization. This is the performance the user can expect to achieve if data are never loaded into a database system

Table 1: Loading Strategies

Data Set: The data set consists of five integer attributes and a total of 107,374,182 tuples. The attributes consist of randomly generated positive integers. In 32-bit integer representation, the binary size of the data set is 2 GB. In HDFS, the data are contained in 32, tab-delimited, text, files of 163 MB each (a total size of 5.1 GB). HDFS block size is set to 256 MB, hence each Map task processes an entire 163 MB split.

3.2 Loading Experiments

The first experiment models a scenario where a user writes a simple job to processes two attributes (a_0, a_1) from an HDFS file. The user filters tuples by a selection predicate on a_0; a_0 lies in the range $[lo - hi]$. The range results in selecting 10% of the data. The user uses a *TabParser*, since the file is tab-delimited, extracts the first and second attribute and filters out the required data. The user re-executes the same job with different ranges. The ranges are randomly selected but maintain 10% selectivity.

The first three strategies from Table 1 immediately copy all the data that need to be processed into the database system. Therefore, the filter operation is pushed entirely into the database layer, and these strategies only differ in how (and when) they reorganize data within the database system.

However, for the invisible loading (IL) strategies, additional data are loaded from the file system into the database system for each job that the user executes, until the data are completely loaded. IL jobs therefore process data from both HDFS and the DBMS: loaded data are processed in the database system and the remaining data are processed within Hadoop (over HDFS).

Figure 2 shows the response time as a sequence of range-selection jobs are executed. The first job represents a *cold* start: caches and memory are cleared and HDFS reads files from disk into memory. The remaining jobs are *hot* executions: all required data are contained within the 12GB main memory.

A complete data pre-load and organization leads to the worst initial response time of about 800 seconds. The baseline MapReduce response time is only 300 seconds. As we move from 800 to 300 seconds, each loading strategy drops a certain amount of loading and reorganization work. First, loading all five columns into 32,

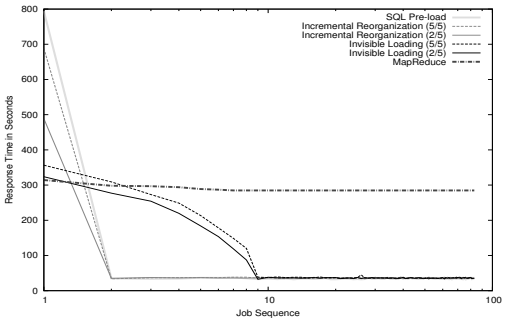

Figure 2: Response time of repeatedly executing selection queries over attributes a_0, a_1.

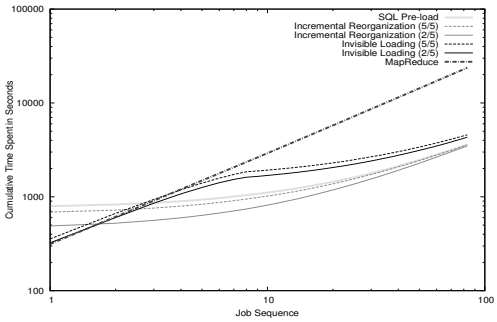

Figure 3: Cumulative cost of repeatedly executing selection queries over attributes a_0, a_1 (Experiment 1).

individually sorted, database partitions ("Incremental Reorganization (5/5)") leads to a query response time of 680 seconds. This 120-second decrease is due to relaxing the constraint of completely sorting the data at one go. Instead, each partition is ordered by a_0[3]. Only after 80 selection queries are issued, is a complete ordering achieved. We gain an additional 200-second decrease in response time if we only load the two processed attributes, a_0, a_1, instead of the entire data set ("Incremental Reorganization (2/5)"). Not only is the size of data smaller, but the number of column alignment operations per partition is reduced from five to two[4].

The goal of invisible loading is that the user should not recognize any major slowdown in the initial query. When only a small fraction of data is loaded from the file system to the database system per job we get closer to this goal. Loading $1/8^{th}$ of the data set, or four partitions per job ("Invisible Loading (5/5)"), drops the initial query response time to 350 seconds. If only two attributes are loaded ("Invisible Loading (2/5)"), the response time becomes 310 seconds, only 3% more than the baseline MapReduce time. By the second query, response time is already lower than MapReduce as $1/8^{th}$ of the data are now processed in the database at a much faster rate than MapReduce. By the eighth query, $7/8^{th}$ of the data are loaded in the database and the last four partitions are processed by MapReduce while being copied into the database. Incremental reorganization happens along with loading. Therefore, like the "Incremental Reorganization" strategies discussed above, a complete ordering is achieved after 80 queries are issued.

Once data are loaded into the database systems, query response time stays within 35 seconds regardless of whether the DBMS is physically reorganizing the data or not. This is because the queries for these experiments are expressed in MapReduce (not SQL), so Hadoop is still used to do the analysis after it reads data from the parallel database. Hadoop, a batch-oriented data processing system, is not well optimized for interactive queries and takes (for our setup) an average of 35 seconds per query just to get it started. Therefore, for these queries at 10% selectivity, the overhead of Hadoop overwhelms the time spent in the database system. A merge operation of two slices consisting of two columns only takes 450 milliseconds in the database system, and a selection operation takes about 750 milliseconds on average. This is why we remove Hadoop and its overhead when comparing reorganization strategies in further experiments (Section 3.3).

3.2.1 The Cumulative Cost of not Loading Data

[3]MonetDB uses a quicksort algorithm to sort columns. Sorting smaller partitions of a column leads to better cache locality than sorting an entire 410MB column at one go.

[4]Note that in addition to the actual columns, a hidden address column is generated.

The short-lived immediate gratification of running MapReduce jobs over the distributed file system has long-term damages. The cumulative advantage of invisible loading is easily deducible from Figure 2.

As seen in Figure 3, the cumulative cost of executing MapReduce jobs over the distributed file system is an order of magnitude higher than any of the five data loading strategies after only eight jobs. This is because each job must scan and parse in a brute-force fashion the entire data set without any making any progress towards preparing the data for future queries. Invisible Loading strategies do not pay the high upfront costs of pre-loading. However, because it must still do some subset of work over the distributed file system until the entire data set is loaded, it incurs a higher cumulative cost than the alternative strategies. The incremental reorganization strategies have approximately the same cumulative effort as preloading the entire dataset over the long run. This result strongly supports our hypothesis that completely reorganizing data at one go has little cumulative benefit over incrementally doing so, especially with analytical-type queries that access large segments of the data. Finally, loading only two out of five attributes always leads to better cumulative effort if none of the three unloaded columns are ever accessed.

3.2.2 Selecting a Different Attribute on Loading

In this experiment, we drop the assumption that none of the other columns are accessed, modeling a more realistic data access pattern. We examine the cost of loading previously unloaded columns, and catching them up to the reorganization that has taken place in the previously loaded columns.

Suppose the user wants to process attributes (a_0, a_2) instead of (a_0, a_1) where a_0 lies in the 10% selectivity range $[lo - hi]$. The three loading strategies that load all attributes into the database whether or not they are accessed by a MapReduce job are not affected by this change, since all columns have already been loaded in the database and are aligned with each other. Partial loading strategies (3 and 5 from Table 1), that have loaded only two out of the five attributes, need not only to load the third attribute, but since the already loaded columns have been incrementally reorganized, additional work must be done to align this third column with the already loaded database columns (See Case 0 in Section 2.3).

We compare the approach described in Section 2.3 with a naive baseline solution that immediately loads and reorganizes the column that has not yet been loaded (instead of loading and reorganizing it incrementally).

Figure 4 illustrates the response time achieved by all strategies when the user selects attributes a_0, a_2 at Job 84 (this number was chosen since the first two columns have been completely reorga-

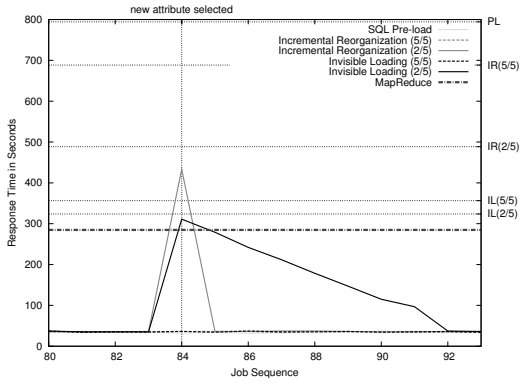

Figure 4: Response time of selecting a different attribute than previously selected. Dotted gridlines indicate the initial load response time.

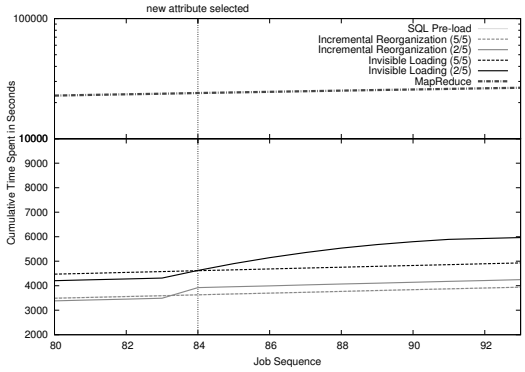

Figure 5: Cumulative cost of selecting a different attribute than previously selected.

nized at this point, so this presents the worst case scenario for catching up the new column with the already loaded columns). The 'load-at-one-go' strategy takes about 433 seconds (it is labeled as "Incremental Reorganization (2/5)" on the figure since the first two columns were loaded using the incremental reorganization technique). The total load and reorganize times spent in the database are 76.1 and 35.1 seconds respectively.

The Invisible Loading strategy follows the same pattern of initially loading two columns. The eighth job includes the additional reorganization cost of 15.2 seconds - this explains the slight upward kink in the graph at job 91.

Figure 5 illustrates the cumulative cost of the different strategies. Up to job 83, partial loading strategies were performing better than there full load counterparts. As soon as a new column needs to be loaded, we are pushed from 35 second response times to at least 290 seconds or the baseline MapReduce response time. Therefore, both partial loading strategies overtake the full loading strategies.

If we examine time spent in the database system, however, the picture changes: loading all five columns into 32 partitions has a total cost of 302 seconds, individually sorting each partition has a total cost of 28.6 seconds (total: 330.6 seconds). This compares with a loading cost of 140.45 seconds and a sorting cost of 22.25 seconds for loading two columns. Loading the additional column costs only 79 seconds and a complete reorganization costs 15.2 seconds (total: 256.9 seconds). Therefore, even without including the complete reorganization costs of reorganizing five attributes, partially loading attributes does not overtake complete-loading.

Finally, Even if we were to load the remaining two attributes, the cumulative rise in the partial loading strategies will not overtake the cumulative cost of MapReduce, which is an order of magnitude more costly than all loading strategies (as seen in Figure 5).

3.3 Reorganization Experiments

The aim of the reorganization experiments is (i) to analyze the behavior of three different data reorganization strategies, cracking (MonetDB's default data reorganization strategy), incremental merge-sort, and presorting, and (ii) to evaluate the benefit of integrating lightweight compression (RLE) with data reorganization as the cardinality of data decreases. Cracking reorganizes column-store data in an automatic, incremental, query-driven fashion [9, 10]. When a selection query is issued against a column, a *cracker column* is created and optimized for the selection range. Data are partitioned, or cracked, across the range and all values within the selection range are contiguous. The cracked pieces contain disjoint key ranges and within each piece data are unsorted. A cracker index maintains information on the key range of each piece. A key property of cracking is that key ranges that are never queried are never partitioned. Heavily querying a specific range with different sub-ranges will create smaller pieces with finer grained key ranges that are better represented in the cracker index.

As explained above, all experiments are executed completely within MonetDB and assume the data have been loaded. Therefore, we do not include the initial load and sort times and focus only on the data reorganization operations that follow.

We generate three data sets, each with a single column of 10^8 integer values. Each data set models a different data cardinality[5]. The first, high cardinality, data set consists of 20% unique values. Values are selected uniformly at random from the range $[0, 2 \times 10^7)$. The second data set consists of 1% unique values selected from the range $[0, 10^6)$. The, third, low cardinality data set has 0.1% unique values, with values selected from the range $[0, 10^5)$. For each of the data reorganization strategies we execute a sequence of 1000 aggregation queries. Each query selects a random range with fixed selectivity, and then sums all the values within the range. We vary query selectivity from 0.1% to 10%.

Figure 6 illustrates a matrix of response time graphs where we vary data cardinality along the columns and selectivity along the rows. We plot moving average response times with a period of four to eliminate noise and better highlight the trends. We will first discuss the general performance of the different data reorganization strategies in comparison to each other without compression. Then, we will evaluate the effect of compression on query response time for presorted and incrementally merge-sorted data.

Presorting: Since data are presorted, the selection operator executes two binary searches to determine the end-point positions of the selected range, and then returns a view of all the tuples within the two positions. The aggregation operator simply scans and sums all tuples in the view. Since no further data reorganization is required, the response time remains constant throughout the sequence of 1000 queries. As selectivity increases, the query response time increases proportionally to the increase in the amount of intermediate result tuples that need to be aggregated.

Cracking: In all graphs, cracking behaves consistently. Query response times initially start at 650 milliseconds and within 10 queries drop an order of magnitude. Cracking swaps tuples inplace such that all tuples that fall within the selection range are contiguous. A view with pointers to the positions of the first and last tuple in the range is then passed to the aggregation operator. The first query causes cracking to copy the data into a new column,

[5]Cardinality refers to the uniqueness of data values contained in a particular column.

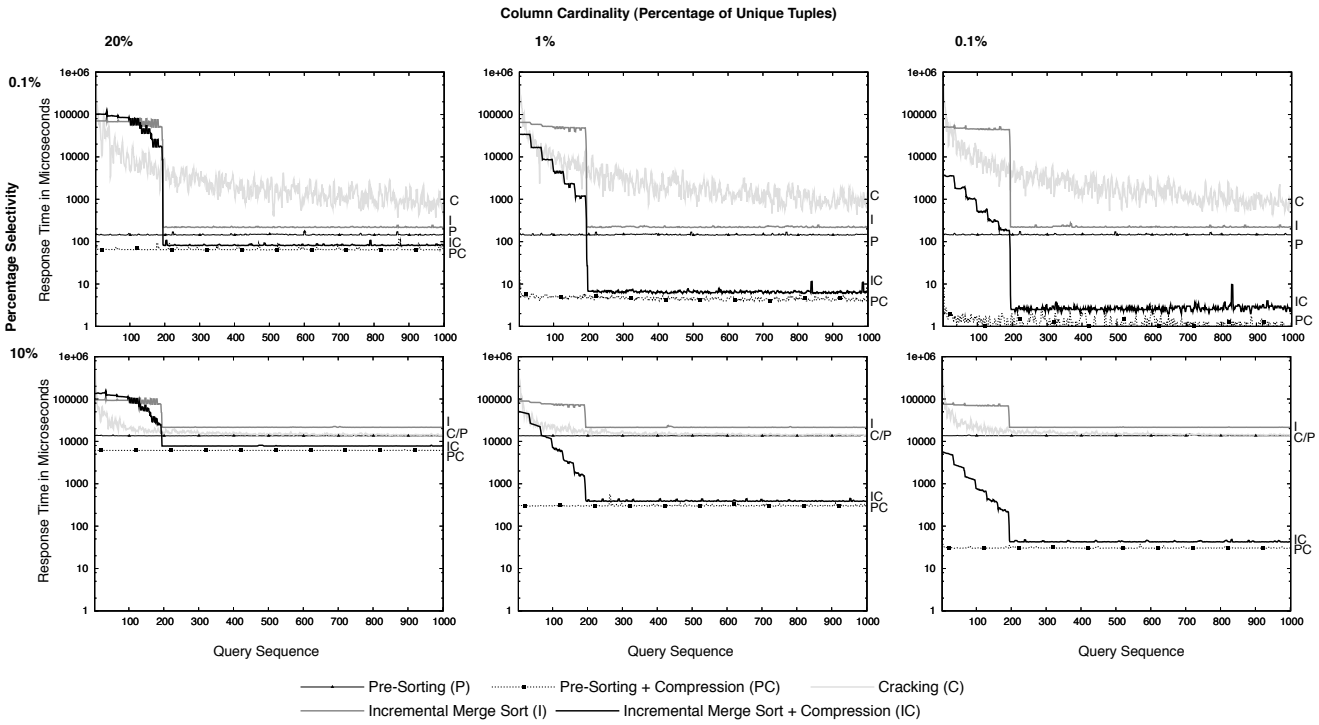

Figure 6: Matrix plot of aggregation queries with selectivity varied vertically across rows and data cardinality varied horizontally across columns. Note: Incremental Merge Sort algorithms manage 64 slices.

the cracker column, and hence it is the most expensive query. Consecutive queries partition the column across the selection range. As cracking continues, tuple movements are usually confined within the boundaries of smaller sub-partitions of the column; hence query response times decrease. As selectivity increases, the overall query response time increases. This is due to the increase in the amount of intermediate result tuples that need to be aggregated.

Incremental Merge Sort: At loading, data are sliced into 64 individually sorted slices. Incremental Merge Sort goes through six phases of 32 merge steps each. Therefore, the entire data set is completely ordered by the 192^{nd} query and an observable drop in query response time occurs then. At each merge step, two slices of roughly 1.5 million tuples each are merged and split. Since, the amount of reorganization work performed at each merge step is roughly equal, query response times remain roughly constant. There is, however, a slight, gradual decrease in response time as the column morphs into slices of tight, disjoint ranges — the selection operator makes use of the range index to only access slices that overlap with the selection range, instead of accessing all slices.

Once data are completely organized, query response time approaches, but never achieves, presorted response times. This, is because we maintain a sliced column organization. Therefore, if the selection range spans multiple slices, the selection operator result-set will consist of non-contiguous tuples and the aggregation operator has to navigate through multiple selection views instead of a single view. Incremental merge sort consistently outperforms cracking due to compression.

3.3.1 Compression and Response Times

Presorting with Compression: We implemented a simple RLE compression operator in MonetDB that associates every value in a column with a count of the number of times it appears consecutively. Since data are sorted, the compressed column will only contain distinct values and associated counts. We also implemented a sum aggregation operator that operates directly on RLE encoded data, without decompression.

With compression, each query (i) searches through less data, (ii) produces a compressed intermediate selection result-set, and (iii) takes advantage of the RLE format when aggregating data (i.e. there is no decompression overhead). Therefore, query performance with pre-sorted and compressed data is better than with pre-sorted and non-compressed data. As the data cardinality decreases, the amount of data that needs to be processed decreases proportionally, hence query response times decrease proportionally.

Incremental Merge Sort with Compression: Figure 6 illustrates two key trends: (i) with high cardinality, 20% unique tuples, integrating compression with our data reorganization strategy is, initially, more costly in terms of query response time, (ii) query response times descend in a "stepwise" fashion.

Before the first query is issued, all 64 slices are individually sorted and run-length encoded. When data cardinality is high, the average run-length of values in each slice is one. Since RLE associates an extra count with every value, the RLE-encoded slices occupy more space than non-encoded slices. Therefore, initially integrating compression with incremental merge sort is expensive on high cardinality data sets. However, as soon as the average run-length starts to increase, due to merging slices, the performance of compressed incremental merge sort improves and outperforms its non-compressed variant. At higher compression ratios, compressed incremental merge sort outperforms cracking, non-compressed incremental merge-sorting and non-compressed presorting throughout the entire sequence of queries. This is because it reorganizes and processes less data. The stepwise descent in query response time occurs at the end of every merge phase. At the end of every phase, the average run length of every value doubles and the

size of the column is nearly halved[6]. Hence, the next merge phase merges exponentially less data and query response time decreases exponentially[7].

Cracking: Cracking is not able to benefit from compression since it does not keep data sorted within each range partition. Hence, the average run-length remains very small in general.

4. SUMMARY

Our experiments indicate that invisible loading poses almost no burden on MapReduce jobs, and makes incremental progress towards optimizing data access for future analysis, yet maintains a cumulatively low price in comparison to performing no data access optimization. We show that Incremental Merge Sort still competes with cracking in main-memory data reorganization of fixed-width columns even though it targets a different data reorganization niche than database cracking (variable-width, compressed and memory-limited data). We also show that integrating lightweight compression into incremental data reorganization improves query performance over low cardinality data sets.

Invisible loading is not a silver bullet to all loading problems. It suffers from data duplication costs. Since, there is no automatic scheme to determine if a particular file has been completely parsed, files are never garbage collected after invisible loading completes. Incremental merge sort spreads data reorganization over many queries. This is beneficial only if most our data is short-lived as we don't waste effort reorganizing the entire data. However, by fine-turning the fractions loaded per IL job and the initial number of sorted slices k used by incremental merge sort, one could speed up or slow down the reorganization process.

5. RELATED WORK

Our research work is influenced by and relates to the following research areas:

In the research area of *Schema Independent querying*, we were influenced by systems like Pig [14] of Yahoo and Swazall [16] of Google. Pig executes over Hadoop and processes files in HDFS without requiring predefined schemas. Instead, schemas are defined at runtime as long as the user provides a parser that could extract tuples from the raw data representation.

In our work, we address a very specific problem within the larger *Schema Evolution* research, that attempts to transparently evolve a database schema to adapt to changes in the modeled reality[6]. Using a column-store, we efficiently evolve a simple schema as we incrementally add attributes. In our system, if a data set was parsed by different parsers, two identical attributes could be considered different by the catalog and maintained separately in the database. However, by using a schema integration system like Clio [8], we could eventually create a unified schema in the database and map it to the different schema representations maintained in the catalog.

The work on *Self-tuning Databases* relates to our work in a complementary fashion. Self-tuning databases monitor query workloads and query execution plans, and create or remove indices and materialized views in response [5]. We could develop similar tools that monitor how data in the file systems are accessed and suggest, which data sets should be migrated to the database.

A recently developed incremental data reorganization technique is *Adaptive Merging* [7]. Similar to cracking, adaptive merging is query driven and optimizes the organization of an index for selection ranges that are heavily queried. Adaptive merging does not guarantee a monotonically decreasing effort spent in data reorganization as it is sensitive to the size of the result-set requested by a query. If a query selects the entire range of the data, adaptive merging performs a complete sort of the data set. It, however, uses an efficient external merge sort.

A recent research work enables users to write SQL queries directly over structured flat files and gradually transfers data from the files to the DBMS [11, 2]. Our system derives schema, loading and reorganizing operations from MapReduce scripts over (less-structured) distributed files. Complementary to our system is Manimal [3], which uses static code analysis to apply relational optimizations to map functions.

6. CONCLUSION

We have designed and evaluated a system that can achieve the immediate gratification of writing MapReduce jobs over distributed file systems, while still making progress towards the longer term benefits of data organization, clustering, and indexing that come with traditional database systems. Invisible Loading gives the illusion that user code is operating directly on the file system, but every time a file is accessed, progress is made towards loading the data into a database system, and then incrementally clustering and indexing the data. Experiments showed that the cumulative performance of invisible loading is very close to the traditional technique of loading all data in advance.

Acknowledgments This work was sponsored by the NSF under grant IIS-0845643.

7. REFERENCES

[1] A. Abouzeid et al. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *VLDB*, 2009.
[2] I. Alagiannis et al. NoDB: efficient query execution on raw data files. In *SIGMOD*, 2012.
[3] M. J. Cafarella and C. Ré. Manimal: relational optimization for data-intensive programs. In *WebDB*, 2010.
[4] R. Chaiken et al. SCOPE: easy and efficient parallel processing of massive data sets. *VLDB*, 2008.
[5] S. Chaudhuri and V. Narasayya. Self-tuning database systems: a decade of progress. In *VLDB*, 2007.
[6] C. A. Curino et al. Graceful database schema evolution: the PRISM workbench. *VLDB*, 2008.
[7] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, 2010.
[8] L. M. Haas et al. Schema AND Data: A Holistic Approach to Mapping, Resolution and Fusion in Information Integration. In *ER*, 2009.
[9] S. Idreos et al. Database cracking. In *CIDR*, 2007.
[10] S. Idreos et al. Self-organizing tuple reconstruction in column-stores. In *SIGMOD*, 2009.
[11] S. Idreos et al. Here are my data files. here are my queries. where are my results? In *CIDR*, 2011.
[12] M.-Y. Iu and W. Zwaenepoel. Hadooptosql: a mapreduce query optimizer. In *EuroSys*, 2010.
[13] MonetDB. Official Homepage. Accessed March 4, 2010. `http://monetdb.cwi.nl/`.
[14] C. Olston et al. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.
[15] A. Pavlo et al. A Comparison of Approaches to Large Scale Data Analysis. In *SIGMOD*, 2009.
[16] R. Pike et al. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.*, 2005.

[6]With non-uniform data distributions, query response time will decrease at every merge phase but not necessarily in a stepwise fashion.

[7]The roughly equal step sizes are due to the log scale but each merge phase takes an order of magnitude less time than the previous phase.