HadoopToSQL

a MapReduce Query Optimizer

Ming-Yee Iu
EPFL
ming-yeechrist.iu@epfl.ch

Willy Zwaenepoel
EPFL
willy.zwaenepoel@epfl.ch

Abstract

MapReduce is a cost-effective way to achieve scalable performance for many log-processing workloads. These workloads typically process their entire dataset. MapReduce can be inefficient, however, when handling business-oriented workloads, especially when these workloads access only a subset of the data.

HadoopToSQL seeks to improve MapReduce performance for the latter class of workloads by transforming MapReduce queries to use the indexing, aggregation and grouping features provided by SQL databases. It statically analyzes the computation performed by the MapReduce queries. The static analysis uses symbolic execution to derive preconditions and postconditions for the map and reduce functions. It then uses this information either to generate input restrictions, which avoid scanning the entire dataset, or to generate equivalent SQL queries, which take advantage of SQL grouping and aggregation features.

We demonstrate the performance of MapReduce queries, when optimized by HadoopToSQL, by both single-node and cluster experiments. HadoopToSQL always improves performance over MapReduce and approximates that of hand-written SQL.

Categories and Subject Descriptors H.2.4 [*Database Management*]: Systems—Parallel databases, query processing

General Terms Languages, Performance

1. Introduction

Programmers are increasingly using MapReduce [5] for performing queries over large datasets. MapReduce transparently handles many of the difficulties of processing data on clusters of commodity hardware, including issues such as fault tolerance, data transfer, and data partitioning. With its increasing popularity, MapReduce has moved from its traditional roots in log-processing to new workloads such as scientific computing [4] and business decision support systems [12]. Programmers have also started using the MapReduce abstraction with storage engines other than cluster file systems [3]. When used in these new contexts, MapReduce has had difficulty taking advantage of potential efficiencies inherent in these new workloads and storage engines [15]. For example, a decision support application might focus on certain subsets of a dataset only. If MapReduce is used with a storage engine that supports features such as indexing or aggregation, the decision support workload can be significantly accelerated by making use of these storage engine features.

HadoopToSQL optimizes MapReduce queries to take advantage of advanced storage engines. It operates on MapReduce queries written for the Hadoop [2] open-source MapReduce implementation. Hadoop queries are written using nor-

```
function map(LogEntry, output):
     output.collect(LogEntry.Country, 1);

function reduce(Country, Iterator, output)
     int sum = 0;
  loop:
     if !Iterator.hasNext() goto end
     Iterator.next();
     sum += 1;
     goto loop
  end:
     output.collect(Country, sum);
```

Figure 1. Pseudocode for a MapReduce query that counts the LogEntries for each country.

```
  SELECT A.Country, COUNT(*)
    FROM LogEntry A
GROUP BY A.Country
```

Figure 2. HadoopToSQL is able to analyze the MapReduce query from Figure 1 and generate this equivalent SQL query.

EuroSys'10, April 13–16, 2010, Paris, France.

mal Java code. Since this code may contain arbitrary computation, optimizing it can be particularly difficult. HadoopToSQL uses static analysis algorithms based on symbolic execution to understand the meaning of these queries. In certain cases, the static analysis performed by HadoopToSQL is powerful enough to translate a MapReduce query entirely to SQL. For example, the MapReduce query in Figure 1 can be translated to the equivalent SQL query in Figure 2. If HadoopToSQL is not capable of generating an equivalent SQL query, it tries to find input restrictions for the query so that the query can take advantage of indexing features of SQL storage engines. HadoopToSQL is notable in that it can optimize MapReduce queries directly as opposed to requiring programmers to write queries in a higher-level query language that is then translated down to MapReduce [1, 14, 21].

By making it possible to run MapReduce code efficiently on SQL databases, HadoopToSQL extends the usefulness of MapReduce to business-oriented workloads where traditional databases still dominate. The greater efficiency of MapReduce when using advanced storage engines also means that MapReduce can be used in single-server settings where it cannot use scalability in order to achieve reasonable performance.

This paper makes the following research contributions:

- We present algorithms for analyzing and understanding MapReduce code.
- We show how this understanding can be used to enhance their performance when used with an advanced storage engine.
- We have implemented and evaluated these algorithms to demonstrate the performance benefits of our approach.

Section 2 provides a more detailed explanation of the motivation behind this paper. In Section 3, we describe the main algorithms used by HadoopToSQL. Section 4 provides more details about the implementation of HadoopToSQL. We evaluate the behavior and performance of HadoopToSQL in Section 5. Then Section 6 discusses related work. Finally, we describe possible extensions to HadoopToSQL in Section 7, and we conclude in Section 8.

2. Background and Motivation

MapReduce is a data processing model designed primarily for large clusters of machines. In a MapReduce cluster, all data is stored as (key, value) pairs. There may be multiple values per key. To perform a query across this data, programmers must define two functions: map and reduce. In Hadoop, map functions take a (key, value) pair as input and output zero or more new (key, value) pairs. Then, the system sorts these new (key, value) pairs. For each key, all the values that correspond to that key are passed as input to the reduce function, which then generates zero or more new (key, value) pairs as output. The final set of (key, value) pairs is saved as the result of the query. Figure 3 shows a conceptual view of how a MapReduce query is executed. Typically, programmers write the code for these two functions using a conventional imperative programming language.

```
function map(key1, value1) : (key2, value2)*
function reduce(key2, value2*) : (key3, value3)*

foreach (key1, value1) in dataset
  temp.addAll(map(key1, value1))
temp.sort()
foreach (key2) in temp.keys()
  result.addAll(reduce(key2, temp[key2]))
```

Figure 3. A conceptual view of how a MapReduce query is executed.

MapReduce is popular because it provides a powerful yet simple-to-understand abstraction that hides many of the difficulties of performing queries on large computer clusters such as dealing with inter-machine communication bottlenecks and machine failure. In practice, MapReduce queries scale well to giant datasets stored across large machine clusters. Since MapReduce is designed to handle failure-prone hardware, it works well with clusters built using commodity hardware, hence providing excellent scalability to large datasets at a reasonable cost.

It is possible to use a traditional declarative query language like SQL or Hive for the same domain [15, 20]. However, queries that need to perform complex computation are ill-suited for declarative query languages but are easily expressed in MapReduce. MapReduce programs are written in conventional imperative programming languages such as Java. Therefore, it is easy to include arbitrary computation such as a complicated AI classifier or mathematical computation. Such computation cannot be expressed directly in declarative query languages but must be programmed externally and then imported into the query language using user-defined functions and stored procedures.

In the research literature, MapReduce has traditionally been used within the context of log-processing workloads [14, 17]. For example, a MapReduce query needs to examine all the log entries of visits to a website to find the most popular web pages on that site. Since these workloads typically require that every record in a dataset be examined, MapReduce is usually paired with a basic cluster file system as a storage engine. All record entries can then easily be streamed off the file system and into the map function.

There are workloads, though, that access only subsets of a dataset. For example, a business might want to analyze their sales in a certain region within a specific date range. For these workloads, streaming through every record in a dataset is extremely inefficient. Indexing the dataset in advance and then using the index to restrict which records are examined is potentially much faster and more efficient. In order to support this possibility, MapReduce needs to be

run using an advanced storage engine that supports indexing, and MapReduce queries need to be analyzed in order to extract information about the subset being accessed.

This analysis is not straight-forward because MapReduce supports arbitrary computation in its map and reduce functions. As a result, any MapReduce query optimizer must be able to analyze arbitrary code in order to extract possible optimizations. HadoopToSQL is designed to optimize Hadoop MapReduce code, in which map and reduce functions are expressed using Java. Since there are at present no advanced storage engines purpose-built for MapReduce, we have targeted our optimizations towards an SQL storage engine.

There already exist possible scenarios where programmers may want to run MapReduce on top of SQL databases. For example, some firms horizontally partition large SQL datasets across many small commodity machines [16]. In such a configuration, queries that access data on only a single machine are fast, but more complex queries that aggregate data across the machines require the use of a distributed SQL database [8, 15] or distributed middleware layer [18, 19]. These firms may choose to use MapReduce for this purpose. Even if a company has an SQL database that fits entirely on a single server, it might decide to write its queries using MapReduce if it believes it will eventually build a MapReduce cluster for data warehousing.

Ultimately, though, HadoopToSQL targets SQL storage engines because they are readily available. The main purpose of HadoopToSQL is to demonstrate that static analysis can be used to better understand MapReduce queries. This understanding can enable performance optimizations for any advanced storage engine.

3. Transformations

The key innovation in HadoopToSQL is a static analysis component that uses symbolic execution to analyze the Java code of a MapReduce query. It transforms queries to make use of SQL's indexing, aggregation, and grouping features. HadoopToSQL offers two algorithms that generate SQL code from MapReduce queries. One algorithm can extract input set restrictions from MapReduce queries, and the other can translate entire MapReduce queries into equivalent SQL queries. Both are intra-procedural algorithms. They function by finding all control flow paths through map and reduce functions, using symbolic execution to determine the behavior of each path, and then mapping this behavior onto possible SQL queries. HadoopToSQL analyzes all MapReduce queries using both techniques. Since translating entire queries into SQL offers more performance benefits than simply finding input set restrictions, that optimization is preferred if both can be applied to a particular query. If none are applicable, then the query is run without optimization.

3.1 Input Set Restrictions in the Map Function

Since database queries tend to be very data-intensive, one of the most important optimizations that can be performed is to reduce the amount of data that needs to be processed. MapReduce queries that operate on only a subset of a dataset can be greatly optimized if HadoopToSQL is able to extract the shape of this subset from the query code and apply this shape as a constraint on the input set of the queries. For example, given a database of a company's sales, a query that analyzes the sales of a certain region only needs to be supplied with data from that region.

Conceptually, HadoopToSQL's algorithm for finding input set restrictions works by tracing through different possible execution paths of the map function. As HadoopToSQL follows the paths of these traces, it records the constraints on variables that need to hold for each trace to occur. If a trace does not result in output being generated, then the trace is ignored. If a trace does result in output being generated, then the input constraints that trigger the trace are included in the input set. There can also be traces that HadoopToSQL cannot fully analyze such as traces with calls to unknown methods. When faced with such imprecise knowledge, HadoopToSQL must make the conservative assumption that this trace generates output. As such, the input constraints that trigger the trace are also included in the input set. The resulting restrictions are not "tight" but do not exclude any data unintentionally.

HadoopToSQL generates these traces by performing a depth-first walk of all paths through the control flow graph of the map function, starting at the entry point and ending at the function exit. It stops traversing along a path upon encountering a loop (which can lead to infinitely long paths) or a statement with unknown side-effects. It then labels that path as not fully analyzable. Statements with unknown side-effects include essentially all method calls, but HadoopToSQL knows about common methods with no side-effects like `String.equals()`, methods of automatically generated entity objects, and methods that are necessary for MapReduce such as `Output.collect()`. This approach to path traversal leads to HadoopToSQL being most effective at finding input constraints in programs that filter their input as early as possible.

To calculate the constraints on variables that need to hold for a trace to occur, HadoopToSQL uses symbolic execution to calculate the preconditions and postconditions of executing the statements of a path. Essentially, each branch on a path becomes a precondition of the path, and each method call and variable assignment to a non-local variable becomes a postcondition. For *each* path that might generate output, HadoopToSQL takes the various preconditions of the path and creates a single precondition expression for the path by ANDing them together. This expression describes the input that trigger the execution of the path. HadoopToSQL then takes these expressions for each path and ORs them all to-

```
function map(Sale, Output):
     if Sale.Region() == "East" goto end
     if Sale.Region() != "North" goto output
     Classification = classify(Sale)
     if Classification.Size() <= 5 goto end
  output:
     Output.collect(
        Classification.SalesCategory(), Sale)
  end:
     return
```

Figure 4. Pseudocode of a map function that analyzes the sales in a certain region.

```
Path 1:
if Sale.Region() == "East"      (branch not taken)
if Sale.Region() != "North"     (branch not taken)
Classification = classify(Sale) (path traversal
                                 aborted)

Path 2:
if Sale.Region() == "East"      (branch not taken)
if Sale.Region() != "North"     (branch is taken)
    goto output
Output.collect(
  Classification.SalesCategory(),
  Sale)

Path 3:
if Sale.Region() == "East"      (branch is taken)
    goto end
```

Figure 5. HadoopToSQL finds three paths through the map function.

gether. This results in a boolean expression that can be used to restrict the input set to the query.

Figure 4 shows an example map function, which will be used to illustrate how the input set restriction algorithm works. The function includes a call to a `classify()` method that potentially contains a complicated algorithm for classifying sales into different categories and sizes. HadoopToSQL first enumerates all paths through the method, truncating paths that include the `classify()` method since the method has unknown side-effects (Figure 5).

HadoopToSQL then uses symbolic execution on each path to determine the preconditions and postconditions of each path. Figure 6 shows the preconditions and postconditions of the two paths from Figure 5.

HadoopToSQL knows that the method `Sale.Region()` has no side-effects because it is an accessor method of an automatically generated entity object. It can thus determine that path 3 does not generate output, that path 2 obviously does generate output, and that path 1 is not fully analyzable. As a result, it uses the input constraints of path 1 and path 2 to generate input set restrictions for the query. The individual

```
Path 1 Preconditions:
   Sale.Region() != "East"
   Sale.Region() == "North"

Path 1 Postconditions:
   Sale.Region()
   (traversal aborted)

Path 2 Preconditions:
   Sale.Region() != "East"
   Sale.Region() != "North"

Path 2 Postconditions:
   Sale.Region()
   Output.collect(
      Classification.SalesCategory(), Sale)

Path 3 Preconditions:
   Sale.Region() == "East"

Path 3 Postconditions:
   Sale.Region()
   (exit function)
```

Figure 6. By using symbolic execution, HadoopToSQL is able to determine the preconditions and postconditions of executing each path through the code.

```
Path 1 Precondition Expression:
   Sale.Region() != "East"
   AND Sale.Region() == "North"

Path 2 Precondition Expression:
   Sale.Region() != "East"
   AND Sale.Region() != "North"

Final Boolean Expression:
      (Sale.Region() != "East"
           AND Sale.Region() == "North")
   OR (Sale.Region() != "East"
           AND Sale.Region() != "North")
```

Figure 7. From the preconditions of each path, HadoopToSQL is able to derive a boolean expression describing the input set restrictions.

preconditions of each path are ANDed together to form input constraints for the path. These expressions are then ORed together, resulting in the final input set restrictions, which may contain redundant terms (Figure 7). The code for reading data into the map function can then be modified to include a WHERE clause with these input set constraints (Figure 8). Although the final WHERE clause may be amenable to further simplification, this task is left to the SQL query engine.

```
ResultSet rs = execute(
      "SELECT * FROM Sale A"
      + " WHERE (A.Region <> 'East' "
      + "          AND A.Region = 'North')"
      + "    OR (A.Region <> 'East' "
      + "          AND A.Region <> 'North')";
while (rs.next())
  Sale s = new Sale(rs);
  apply map to s
```

Figure 8. Pseudocode for how the input set constraint appears in the WHERE clause of an SQL query for feeding data into a map function.

3.2 Complete Translation to SQL

HadoopToSQL's second transformation algorithm can translate entire MapReduce queries into a single SQL query. Such a query can be more efficient than a normal MapReduce query by reading only the fields of a record that are used by the query. It can also make use of aggregation optimizations in SQL databases. For example, a query might divide its data into a large number of categories based on whether the value of a field fits within certain ranges. It might then calculate aggregates for each category. If a database has sorted its dataset by the same field, it can calculate these aggregates with a single pass through the data. Finally, a query that is fully translated to SQL can also make use of input constraints.

Unfortunately, since the query model supported by MapReduce cannot be mapped directly onto the SQL query model, this transformation is only feasible for certain classes of MapReduce queries. HadoopToSQL can only translate MapReduce queries fulfilling these general properties into SQL queries:

For the map function:

- Any execution of the map function can emit at most one (key, value) pair.
- The function can make arbitrary use of `if` statements but it cannot contain any loops.
- The function can only use operators and functions that exist in SQL.
- The function can create and modify only local variables. These variables must have types that are compatible with SQL.

For the reduce function:

- The reduce function must emit exactly one (key, value) pair.
- The (key, value) pair output by the function must use same key that is used for its input (key, value) pairs.
- The function can only use operators and functions that exist in SQL.
- The function can create and modify only local variables. These variable must have types that are compatible with SQL.
- The reduce function should either be the identity function, or it should iterate over its input values and compute some sort of aggregation that is compatible with SQL.

Most of these properties are the result of the inherent restrictions of the SQL query syntax and are not due to inflexibility in the transformation algorithm. For example, an SQL query can output at most one output row for each input row processed, so for a map function to be translated into an SQL query, it too can only output at most one (key, value) pair for each input (key, value) pair.

Conceptually, the transformation that HadoopToSQL performs is that it tries to fill in a stencil of a SELECT...FROM...WHERE...GROUP BY query based on the behavior of the map and reduce functions. HadoopToSQL extracts an input restriction from the map function, and uses it as the WHERE clause of the SQL query. The (key, value) pair generated by the map function is used as the SELECT clause of the SQL query. If the reduce function calculates an aggregation, then a GROUP BY clause is added to the query with a grouping based on the key, and the SELECT clause is modified to aggregate the values computed in the map.

The analysis of the map function is performed using the same method as described in Section 3.1, but HadoopToSQL needs to fully understand the behavior of the code instead of merely calculating a conservative approximation. Once the map code is broken up into paths and after the preconditions and postconditions have been calculated through symbolic execution, HadoopToSQL can use the path preconditions to compute an expression for the WHERE clause. There should be no ambiguous operations in the preconditions, so the resulting input set restrictions are exact. Since the data being output by the map function are encoded in the path postconditions, HadoopToSQL can simply extract the expressions being output and use them in the SELECT clause.

For example, consider the map and reduce functions in Figure 9. The program divides sales into two categories—one category for the "North" region and one category for the others—and calculates the total commission on sales for each category. There are two paths through the map function, both of which generate output. Figure 10 shows the preconditions and postconditions for the two paths, and it shows how to derive a single SELECT clause that is equivalent to the two paths.

The extraction of aggregation information from the reduce function is more involved because the reduce function must use a loop to iterate over its input. The loop in the function is found by using a strongly-connected components algorithm. All the paths through this loop are enumerated and the preconditions and postconditions for each path are calculated using symbolic execution (Figure 11). HadoopToSQL has a series of template patterns that describe how vari-

```
function map(Sale, Output):
     if Sale.Region() != "North" goto L1
     Output.collect("North", Sale.Commission())
     goto mapend
  L1:
     Output.collect("NotNorth", Sale.Commission);
  mapend:
     return

function reduce(key, Iterator, Output):
     sum = 0
  loop:
     if !Iterator.hasNext() goto reduceend
     sum = sum + Iterator.next()
     goto loop
  reduceend:
     Output.collect(key, sum)
     return
```

Figure 9. Pseudocode for a MapReduce program that can be translated completely into SQL.

```
Path 1 Preconditions:
     Sale.Region() == "North"
Path 1 Postconditions:
     Output.collect("North", Sale.Commission())

Path 2 Preconditions:
     Sale.Region() != "North"
Path 2 Postconditions:
     Output.collect("NotNorth", Sale.Commission())

SELECT CASE WHEN A.Region = "North" THEN "North"
            ELSE "NotNorth" END,
       A.Commission
  FROM Sale A
```

Figure 10. The SELECT clause of an SQL query can be computed based on the preconditions and postconditions of the paths through the map function. This SELECT clause calculates only two fields: one for the map function's key and one for the map function's value. Because the key differs based on the input, a CASE statement is required.

ous SQL aggregation operations are expressed as a loop's preconditions and postconditions (Figure 12). By matching these templates against the loop, it is able to identify which SQL aggregation is being used. HadoopToSQL currently has templates for recognizing SQL's SUM, MIN, and MAX aggregation operations. The templates look for loops that iterate over a collection and that collect a result in a single variable.

HadoopToSQL can then analyze the non-loop code of the reduce function by again using symbolic execution to calculate path preconditions and postconditions. The symbolic execution engine treats the loop as a single statement that calculates an aggregation. If the rest of the reduce code satisfies all the reduce function properties described earlier in this section, then the key is added as a GROUP BY to the query, and aggregation operations are applied to the values in the SELECT clause. Since MapReduce results appear in sorted order, HadoopToSQL also adds an ORDER BY clause to the final query.

```
Path 1 Preconditions:
     !Iterator.hasNext()
Path 1 Postconditions:
     exit loop

Path 2 Preconditions:
     Iterator.hasNext()
Path 2 Postconditions:
     Iterator.next()
     sum = sum + Iterator.next()
```

Figure 11. The loop of the reduce function in Figure 9 has these preconditions and postconditions, which indicate that it calculates a SUM() aggregation.

```
Path i Preconditions:
     !Iterator.hasNext()
     ...
Path i Postconditions:
     exit loop

Path n Preconditions:
     Iterator.hasNext()
     ...
Path n Postconditions:
     Iterator.next()
     sum = sum + expression
```

Figure 12. Template pattern for identifying a loop as a SUM aggregation. In the template patterns for MAX and MIN aggregation, the addition operation is replaced by a max and min operation respectively.

So in our example, if the loop is found to match a template for a SUM aggregation, then the loop of the reduce function is replaced by a single statement summarizing the effect of the loop (Figure 13). Symbolic execution is then applied to the entire reduce function to calculate postconditions, which allows us to discover the reduce function encodes a GROUP BY query (Figure 14). The SELECT clause used to calculate the outputted key and value of the map function can then be merged into a GROUP BY stencil to produce a final SQL query for the combined map and reduce functions (Figure 15). The final query may contain expressions that can be further simplified, but this task is left to the SQL query engine.

```
function reduce(key, Iterator, Output):
     sum = 0
 loop:
     sum += SUM(Iterator values)
 reduceend:
     Output.collect(key, sum)
     return
```

Figure 13. The loop inside the reduce function is replaced by a statement summarizing the effect of the loop.

```
Path Postconditions:
     Output.collect(key, 0 + SUM(value))

Matching GROUP BY stencil:
  SELECT key, SUM(value)
    FROM ...
GROUP BY key
ORDER BY key
```

Figure 14. If the postconditions for the non-loop portions of the reduce function show that the function satisfies the needed properties, the SQL query can be converted to use a GROUP BY and aggregation.

4. Implementation Details

The HadoopToSQL system consists of a static analysis component and a runtime component. The static analysis component is applied to the Java bytecode of a MapReduce query. It attempts to apply different transformations to the code to try to find an efficient way to execute the code on an SQL database. The runtime component provides a simple object-relational mapping tool to simplify access to database entities. It includes runtime libraries for mapping the SQL data model to fit the MapReduce data model.

4.1 Static Analysis Component

The static analysis component of HadoopToSQL is implemented as a bytecode rewriter. It is able to take a compiled MapReduce program generated by the Java compiler and analyze it to find ways to run it efficiently on an SQL database.

Although the HadoopToSQL bytecode rewriter accepts Java bytecode as input, its internal processing is actually based on a representation called Jimple, a three-address code version of Java bytecode. It uses the SOOT framework [23] from Sable to transform Java bytecode to this representation. Raw Java bytecode is difficult to process because of its large instruction set and the need to keep track of the state of the operand stack. In Jimple, there is no operand stack. There are only local variables, meaning that HadoopToSQL can use one consistent abstraction for working with values.

The static analysis component outputs a data structure that contains descriptions of how various map and reduce functions can be translated to SQL. The HadoopToSQL runtime component can then query this data structure when deciding on a procedure for executing MapReduce queries. Because of HadoopToSQL's design as a bytecode rewriter, it can be added to the toolchain as an independent module, with no changes needed to existing IDEs, compilers, virtual machines, or other such tools.

```
Output of the map function:
SELECT
       /* Key */
       CASE
         WHEN A.Region = "North" THEN "North"
         ELSE "NotNorth" END,
       /* Value */
       A.Commission
  FROM Sale A
 WHERE /* Input restriction */
       A.Region = "North"
       OR A.Region <> "North"

Stencil for the reduce function's GROUP BY:
  SELECT key, SUM(value)
    FROM ...
   WHERE input restriction
GROUP BY key
ORDER BY key

Final SQL query:
  SELECT CASE
         WHEN A.Region = "North" THEN "North"
         ELSE "NotNorth" END,
         SUM(A.Commission)
    FROM Sale A
   WHERE A.Region = "North"
         OR A.Region <> "North"
GROUP BY CASE
         WHEN A.Region = "North" THEN "North"
         ELSE "NotNorth" END
ORDER BY CASE
         WHEN A.Region = "North" THEN "North"
         ELSE "NotNorth" END
```

Figure 15. Merging the SELECT clause of the map function with the GROUP BY stencil of the reduce function results in the final SQL query.

4.2 Runtime Component

HadoopToSQL contains various runtime libraries for allowing an SQL storage model to mix with a MapReduce approach to data.

For example, with MapReduce, data records are typically stored as text in files, whereas in SQL, data records are stored as relations in tables. Neither storage representation is particularly convenient for programmers, who pre-

fer mapping these representations to an object representation inside their programs. HadoopToSQL includes a simple object-relational mapping (ORM) tool that can perform this mapping of either text or relations to entity objects. This hides the differences between the two storage models and provides a more convenient interface for programmers. Programmers provide an XML description of a schema, and the ORM tool creates corresponding entity object classes as well as code for reading these objects from either a text file or from an SQL database. Programmers can then express their MapReduce programs in terms of manipulating these objects instead of needing to write code for parsing text input or for querying databases.

The Hadoop implementation of MapReduce provides a `FileInputFormat` object for reading lines of text from files. The HadoopToSQL library provides alternate objects that can read their data from either databases or files and that can return ORM entity objects instead of lines of text. To switch between using an SQL database as storage engine as opposed to a MapReduce distributed file system, programmers merely have to change the configuration information of their MapReduce queries to use the HadoopToSQL libraries for managing their input.

5. Experimental Evaluation

To evaluate HadoopToSQL, we have run some single-server experiments and one distributed experiment. The single server experiments allow us to compare the performance of SQL queries generated by HadoopToSQL directly to hand-written SQL queries run on a standard single-server database. The distributed experiment verifies that the performance benefits of HadoopToSQL still hold on a cluster. For all of the experiments, data is loaded into databases and indexed before the experiments are run.

5.1 Single-Server Experiments

The single-server experiments are run on a dual-processor Pentium IV Xeon machine with 4 GB of RAM running Linux, OpenJDK 1.6, Hadoop 0.20, and PostgreSQL 8.3. Hadoop is configured for stand-alone operation, with its input and output files stored on the local disk.

5.1.1 Stock benchmark

To illustrate the behavior of HadoopToSQL, we have created a benchmark involving a database of synthetic stock market prices. The database consists of 10,000 different stocks. For each stock, the database tracks the daily closing price and trading volume. To examine the effect of database size, the number of days of historical stock data can be varied between 500 and 3,500 days. When stored in an SQL database, the historical data uses the stock symbol and date as a primary key. A text dump of a database with 3,500 days of data is 970 MB in size.

Our benchmark executes a query that calculates sums of 15 different stocks over a period of five months. This query is inspired by the type of computation involved in calculating stock market indices like the Dow Jones Industrial Average. We measure the performance of this query in the following configurations:

- Hand-written SQL
- MapReduce running on a single machine
- MapReduce running on SQL without any optimizations by HadoopToSQL
- MapReduce running on SQL with input set restrictions calculated by HadoopToSQL
- MapReduce running on SQL with a full translation by HadoopToSQL to SQL

Figure 16 shows the query times for each of these variations. Each data point is the average of 10 query executions with each execution using a random set of 15 stocks. Both regular MapReduce and MapReduce on SQL without optimizations exhibit increasing query time as the database size increases. This is caused by the fact that both variations must scan through the entire database in order to find the stocks and days relevant to the query. As the database size increases, the queries must examine more data as well. For example, given 3,500 days of stock data, a full scan of the dataset needs to examine 35 million records. The performance of MapReduce on SQL without optimizations is approximately 50% worse than that of regular MapReduce. This is due to the fact that it must perform a table scan of an SQL database instead of reading its data from text files like regular MapReduce. Although an SQL database can theoretically store its data in a more compact representation than the textual representation used in MapReduce, SQL databases are rarely optimized for this sort of access pattern, so they do not necessarily fill disk blocks to the maximum extent or arrange data sequentially on disk. By contrast, linear traversals of files is a well-optimized access pattern for operating systems.

The granularity of the y-axis in Figure 16 hides significant detail, so Figure 17 is included in this paper to show an enlarged view of the same data. Hand-written SQL, the restricted input set configuration, and the full translation configuration are all able to indicate to the underlying database that they only want a subset of the data. As such, the SQL database is able to make use of underlying indices to ignore the extra data in the database, meaning that these queries only need to examine approximately 2,000 records. Although all three configurations process the same number of records, the full translation configuration spends time creating XML configuration files, starting a Hadoop MapReduce engine, sending the configuration information to the MapReduce engine, and other non-query-related overhead. This configuration is thus half a second slower than hand-written SQL despite the fact that both configurations execute essentially the same SQL query against the database. Due to

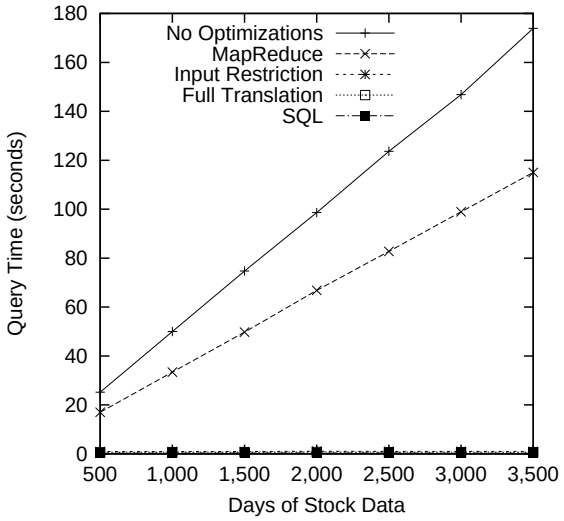

Figure 16. Query time on stock benchmark as database size increases.

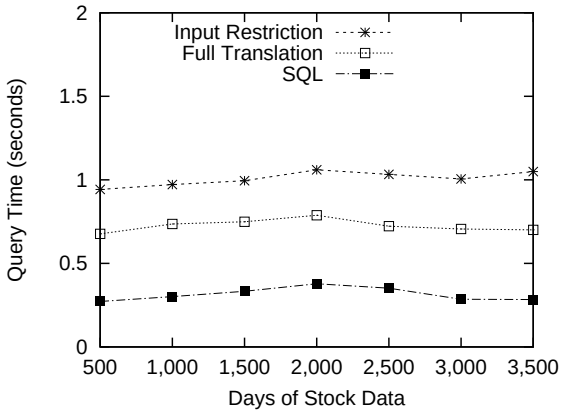

Figure 17. Query time on stock benchmark as database size increases (with zoomed y-axis).

the short running time of the query, Figure 17 exaggerates the size of this overhead. The restricted input set configuration runs a full execution of MapReduce, applying the map and reduce functions to its data, so it has the worst performance of the three.

This benchmark shows the importance of using indices in order to extract the best performance for MapReduce queries running on an SQL database. The MapReduce query model has no notion of indices since the information about which data is used by a query is encoded in the program code itself, which cannot normally be inspected by a MapReduce runtime. The program analysis performed by HadoopToSQL is able to extract this information and hence take advantage of the indices available in SQL.

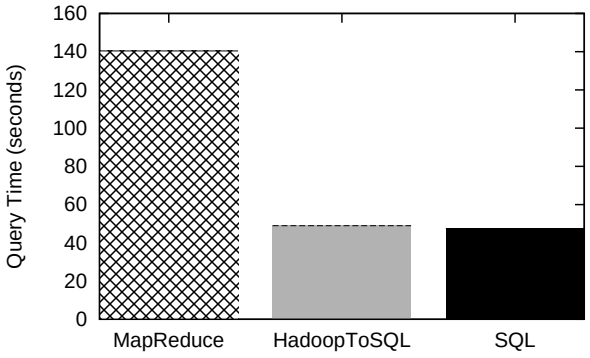

Figure 18. Query time for TPC-H query Q1.

5.1.2 TPC-H

TPC-H [22] is a standard SQL database benchmark for decision-support workloads. We include this benchmark in our experiments because it allows for easy comparison of MapReduce results with SQL results and because it provides an interesting business-oriented workload. Nonetheless, the results must be interpreted with caution because a direct translation of TPC-H queries to MapReduce does not necessarily reflect how such queries would be written and how the schema would be designed if the benchmark specifically targeted a MapReduce query model.

For this experiment, we choose to examine queries Q1 and Q3 of TPC-H, which map well to MapReduce and have non-trivial running times. The benchmark is configured with a TPC-H scale factor of one, resulting in a dataset of approximately 1,100 MB in size. For our experiment, we use random query parameters as specified by TPC-H.

Query Q1 scans a single table of order line items within a certain date range and calculates aggregates for different categories. Figure 18 shows the query results for query Q1. Similar to the stock benchmark, HadoopToSQL is able to extract an equivalent SQL query from the MapReduce code. As a result, the translated query is able to make use of database indices, resulting in much better performance than regular MapReduce, which must scan the entire contents of the text file of line order items. The translated query also exhibits performance that is almost as good as hand-written SQL.

Unlike query Q1, query Q3 involves a database join. Query Q3 examines customer, order, and order line item information to determine the 10 highest-valued orders with certain characteristics and that have not been shipped. It needs to join the customer, order, and line item entities in computing its result. Since MapReduce does not have any built-in support for joins (joins are fundamentally slow operations when applied to data in a cluster), programmers normally structure their data differently if they intend to query it with MapReduce. In particular, programmers denormalize their data in advance to avoid the poor performance of joins

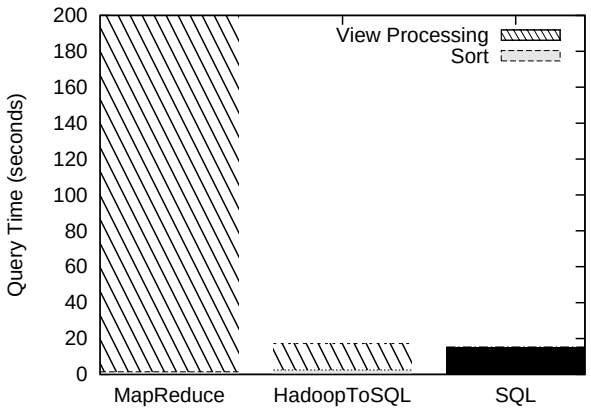

Figure 19. Query time for TPC-H query Q3 when the Customer, Order, and LineItem tables are joined in advance.

in MapReduce. To reflect this fact, we use different data layouts for each configuration.

The SQL query is run using separate tables for each of the three entities. HadoopToSQL also stores the data in three separate tables, but the tables are joined at runtime and presented to the map function as a single table, much like an SQL view. For regular MapReduce, we run the query against a file with the three entities joined in advance. Our coding of the MapReduce and HadoopToSQL versions of the queries have one potential inefficiency as compared to the SQL query. TPC-H specifies that for query Q3, only the top 10 results are needed. In our MapReduce and HadoopToSQL versions of the queries, the top 10 results are found by calculating all the results and sorting them—it is potentially more efficient to calculate the top 10 results directly.

Figure 19 shows the execution times for TPC-H Q3. Regular MapReduce is significantly slower than both HadoopToSQL and SQL. This is due to the fact that it must scan through all the records of the dataset without being able to restrict itself to only those orders that have not yet shipped and that satisfy the expected characteristics. HadoopToSQL is able to extract useful input constraints from the query and is hence able to achieve comparable performance to the hand-written SQL version of the query.

For completeness, we have also created versions of the MapReduce and HadoopToSQL queries that can operate on a dataset that has not been denormalized. This requires that the Customer, Order, and LineItem records be joined during query execution, which we emulate using multiple MapReduce steps. Our version of query Q3 uses six different applications of MapReduce to calculate its result: three stages filter and reformat input records, two stages join these records together and aggregate the results, while a final stage is used to sort the results.

Figure 20 shows the resulting execution times. The individual times of each of the six MapReduce stages are shown

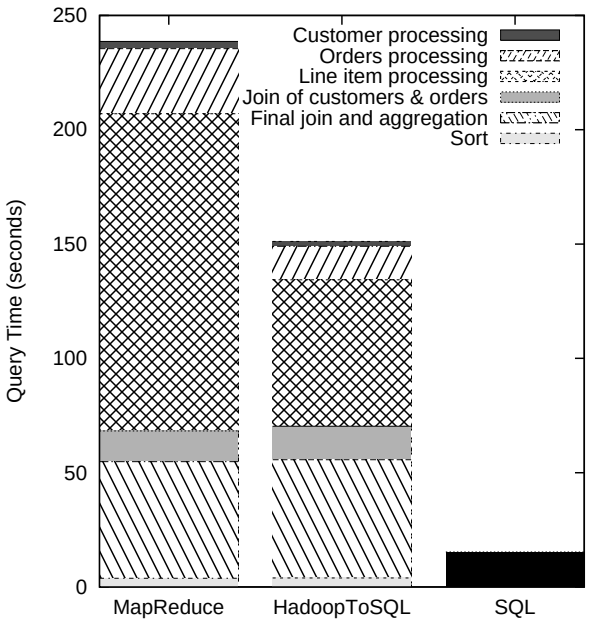

Figure 20. Query time for TPC-H query Q3 when the joins of the Customer, Order, and LineItem tables are performed by MapReduce.

where applicable. HadoopToSQL is able to improve the performance of the MapReduce query when it is run on an SQL database, but it is not able to achieve performance comparable to that of a hand-written SQL query (unlike with the denormalized version of the query). The problem is that HadoopToSQL only optimizes within a single application of MapReduce. HadoopToSQL is not able to optimize across the six MapReduce stages of this version of the query.

The TPC-H benchmark shows that HadoopToSQL can be used to improve the performance of real queries. For a MapReduce query to achieve comparable performance to SQL on a single server, it is important to extract as many input constraints on the query as possible so as to reduce the amount of data that needs to be processed. HadoopToSQL is effective at extracting such constraints from within a single application of MapReduce, but it is currently not able to extract constraints across multiple MapReduce stages.

5.2 Distributed Behavior

To evaluate whether the benefits of HadoopToSQL still hold in the distributed case, where there is additional communication and coordination overhead, we have created an experiment involving a small cluster of machines. We use the Selection Task from the paper of Pavlo et al. [15]. This task involves scanning a list of PageRanks for the URLs of different web pages. The task outputs those URLs with a PageRank greater than the parameter 10.

Configured using the default parameters, the data generation code from the paper generates 5.6M ranking records

per data node in the cluster, for a total size of about 300MB per node. For the SQL and HadoopToSQL configurations, the dataset is divided into equal-sized partitions. An SQL database is running on each data node, and each data node stores one of these partitions in its database. The records are stored with indices for URLs and for PageRank. For the MapReduce configuration, the dataset is stored in the Hadoop distributed file system, which automatically distributes the data among the data nodes.

Our experiment uses 10 data nodes running on Amazon's Elastic Compute Cloud (EC2). We use a "small " EC2 instance for each node, which are configured with a single virtual core, 1.7GB of RAM, and 160GB of local disk space. We use two additional nodes to run the Name Node and Job Tracker servers needed by Hadoop for tracking distributed file system metadata and coordinating MapReduce jobs. All the machines run Fedora 8, Hadoop 0.20, Java 1.6, and PostgreSQL 8.2.

The selection task can be completed by MapReduce in this way:

- During the map phase, each data node scans through ranking records, outputs those URLs that satisfy the query, and stores the results into the distributed file system.
- After the map phase has executed, the result of the query has been computed but is stored in multiple files distributed throughout the cluster.
- The reduce phase transfers these files to a single node, which combines them into a single sorted file.

Although HadoopToSQL can translate MapReduce programs into SQL queries, it currently does not contain code for running SQL queries on a cluster of SQL machines. As a result, for this experiment, HadoopToSQL is only able to use its transformations to find input set restrictions. To estimate the performance of this task on an SQL database, we use a small program that emulates the behavior of a distributed SQL database as we do not have access to one. This program launches 10 threads that each queries one of the databases. The results are then transferred back to this program and stored to disk in no particular order.

Figure 21 shows the results of running the benchmark. Each data point is an average of three benchmark runs. For MapReduce and HadoopToSQL, we show two results. The foreground bar shows the time needed to run the map phase of the MapReduce job only. The background bar includes the time needed to also run a reduce phase. Depending on how the user intends to use the data, they may or may not require the extra processing performed by the reduce phase.

In this experiment, HadoopToSQL is able to find an input set restriction successfully, resulting in better performance than MapReduce. Both HadoopToSQL and SQL are able to restrict their processing to only the 300,000 records of data per node that satisfied the query. HadoopToSQL's map phase

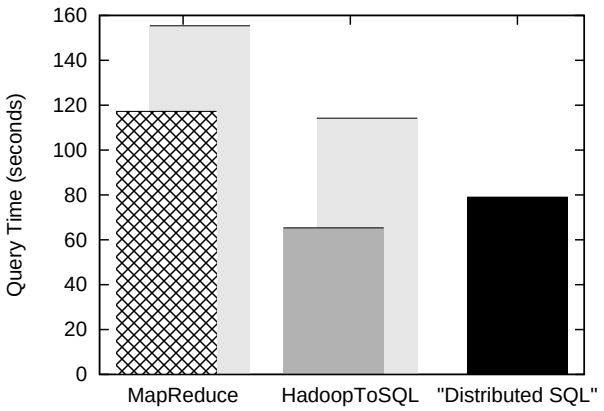

Figure 21. In this graph of execution time for the Selection Task, the results for MapReduce and HadoopToSQL are shown using two bars—the foreground bar shows the results of the map phase only, whereas the background bar includes the time of a reduce phase for gathering the results into single file.

is significantly faster than MapReduce's map phase, but the improvement is less when the reduce phase is included. This occurs because input set restrictions only help the map phase of a query and do not shorten the reduce phase. Although the total time of the HadoopToSQL query is longer than the estimated time for our SQL query, the map phase of the HadoopToSQL query takes less time than the SQL query. This occurs because our SQL program gathers all the query results on a single node, resulting in a potential communication and disk bottleneck on that one node. Although the results of the MapReduce and HadoopToSQL queries are known after the map phase, the query results are stored in multiple files spread out among the data nodes. These results are only merged together into a single file during the reduce phase. Because the reduce phase of a MapReduce program starts while the map phase is still running, it is not possible to determine the actual duration of a reduce phase from the graph. In fact, the reduce phase of the HadoopToSQL query has a shorter overlap with the map phase than the MapReduce query due to the shorter runtime of the HadoopToSQL query's map phase.

6. Related Work

Writing queries with HadoopToSQL is reminiscent of Object-Relational Mapping (ORM) tools [7] like Hibernate or Ruby on Rails. Programmers write code in an object-oriented language, and that code is automatically translated into SQL underneath. In fact, HadoopToSQL uses a simple ORM tool to provide a thin object-oriented layer that hides the difference between the relational storage of SQL databases and the text-oriented storage typically used with MapReduce.

ORM tools do not analyze program code, so they cannot handle the types of queries that HadoopToSQL handles.

Wiedermann, Ibrahim, and Cook [24, 25] have demonstrated a system that can analyze program code in order to generate database queries. They perform interprocedural source-code analysis in order to optimize a program's database access using input-set restrictions. Given a Java program that iterates over all the entities of a certain type in a database, their system uses attribute grammars and abstract interpretation to discover which subset of the entities are being accessed, as well as which related entities are accessed. Their system then restricts the iteration to only cover the database subset which is accessed and prefetches the related entities. Their research focuses on an OODB [13] style query model, so their system cannot be used directly to optimize Hadoop MapReduce queries, even though it optimizes code written in Java.

We have previously developed two systems, Queryll [10] and JReq [11] for translating database queries written in Java code in a restricted loop-based format to SQL. We adapted many of the techniques we learned in developing those systems for use with HadoopToSQL, but we also developed new algorithms for HadoopToSQL.

There are several query languages built on top of MapReduce such as Hive [21] or PigLatin [14]. These query languages are much less verbose than regular MapReduce, and their restricted structure can be analyzed with conventional techniques. Unfortunately, a programmer loses many of the benefits of MapReduce by using such query languages. One of the main advantages of MapReduce is that programmers can perform arbitrary computation at data nodes. This computation can save communication bandwidth by aggressively filtering, compressing, and transforming data before the data is transfered. The restricted syntax of query languages built on top of MapReduce is not rich enough to express such complex algorithms.

Recently, there has been considerable debate about the relative merits of SQL-based approaches to querying data stored on a cluster of machines versus MapReduce-based approaches [6, 15, 20]. The two approaches show different strengths and weaknesses in areas such as scalability, fault tolerance, performance, and flexibility. Research such as HadoopToSQL and HadoopDB [1] demonstrate that it is possible to blend both approaches to build systems with the strengths of both. HadoopDB is another system that runs MapReduce queries over an SQL storage engine. Unlike HadoopToSQL, HadoopDB requires its queries to be written in a Hive-derived query language. This requirement is important because although HadoopDB may generate code that uses MapReduce and SQL, it is not able to directly optimize MapReduce code. As a result, this approach has the same limitations as Hive—although the queries are easier to analyze and optimize, they are not sufficiently expressive to describe complex performance-enhancing algorithms.

DryadLINQ[26] is a query language that is expressive enough to describe arbitrary computation. DryadLINQ adapts the LINQ query language for the Dryad [9] distributed execution engine. The DryadLINQ researchers are also studying how to adapt DryadLINQ to support advanced storage engines.

7. Extensions

Although HadoopToSQL is already very powerful, there are many ways to extend the work to increase its usefulness. In particular, the core static analysis algorithms can be made less restrictive, a traditional distributed database query optimizer can be added, and an advanced storage engine can be designed specifically for MapReduce.

HadoopToSQL's symbolic execution currently halts when it encounters loops or outside functions while searching for input set restrictions. Although exploring loops and outside functions can lead to an exponential explosion of paths, sometimes this explosion is manageable, so HadoopToSQL could undertake limited explorations of loops and outside functions. Loops and outside functions can also be separately analyzed in advance of path traversals. For example, a system can check if a function is free of side-effects by verifying that it neither modifies any non-local variables nor calls any other function with side-effects. Calls to these functions can then be used in HadoopToSQL's symbolic execution. The return value of the function may be ambiguous, but symbolic execution can handle such ambiguity. Alternately, other researchers have successfully used other approaches such as attribute grammars for finding input set restrictions [25] .

HadoopToSQL also currently lacks the ability to optimize across multiple instances of MapReduce. Complex MapReduce programs sometimes consist of multiple stages or instances of MapReduce chained together. The static analysis of HadoopToSQL allows it understand the operations performed by individual instances of MapReduce but is not useful in analyzing the relationship between instances. To solve this problem, HadoopToSQL would first need to provide programmers a mechanism to describe the flow of data between different MapReduce instances. The system could then combine this information with its analysis of individual MapReduce stages to build a query plan describing the complete computation. Once a query plan is built, a traditional database query plan optimizer can be used to rearrange elements of the plan to produce a more optimal execution. HadoopDB [1] operates directly on MapReduce query plans generated from the Hive query language, and it demonstrates some of the possibilities of applying traditional database query optimization techniques to MapReduce.

Finally, additional performance gains can be achieved by building advanced storage engines specifically for use with MapReduce instead of relying on SQL databases. As we noted in our experiments, traditional databases arrange their

data to allow for random-access and updates instead of linear table scans. Therefore, on workloads that need to process their entire dataset, using these databases is slower than using files stored in a MapReduce distributed file system. A purpose-built storage engine for MapReduce could arrange its data in compressed flat files to allow for optimal linear table scans but also provide indices for random-access. An advanced storage engine purpose-built for MapReduce could also take advantage of the fact that intermediate MapReduce results are always saved on disk by reusing these intermediate results for other queries that calculate the same values or subsets of the same values.

8. Conclusion

HadoopToSQL uses static analysis algorithms based on symbolic execution to understand MapReduce queries and optimize them for advanced storage engines. On workloads that access only a subset of a dataset, the performance of MapReduce queries can be significantly improved through such optimizations.

Our evaluation has shown that HadoopToSQL is indeed able to understand MapReduce queries and optimize them for an SQL storage engine. Because the resulting queries are able to take advantage of SQL facilities such as indices, the queries are able to execute much more efficiently using an SQL database than using traditional MapReduce files. In many cases, HadoopToSQL is able to generate SQL code from MapReduce programs whose performance approximates that of hand-written SQL.

HadoopToSQL currently has difficulty analyzing MapReduce programs with loops and unknown method calls, and it is also unable to analyze across multiple MapReduce instances. These limitations can be addressed by adding special analysis algorithms specifically for loops and function calls, and by incorporating a traditional distributed database query optimizer into HadoopToSQL.

Acknowledgments

We would like to thank our EuroSys shepherd Robert Grimm and the EuroSys program committee for their comments and help in improving this paper.

References

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[2] Apache Software Foundation. Hadoop. `http://hadoop.apache.org/core/`.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association.

[4] S. Chen and S. W. Schlosser. Map-Reduce meets wider varieties of applications. Technical Report IRP-TR-08-05, Pittsburgh, USA, 2008. Intel Research Pittsburgh.

[5] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[6] J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[7] L. DeMichiel and M. Keith. JSR 220: Enterprise JavaBeans 3.0. `http://www.jcp.org/en/jsr/detail?id=220`, May 11 2006.

[8] D. J. DeWitt, S. Ghanderaizadeh, and D. Schneider. A performance analysis of the gamma database machine. In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 350–360, New York, NY, USA, 1988. ACM.

[9] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[10] M.-Y. Iu and W. Zwaenepoel. Queryll: Java database queries through bytecode rewriting. In M. van Steen and M. Henning, editors, *Middleware*, volume 4290 of *Lecture Notes in Computer Science*, pages 201–218. Springer, 2006.

[11] M.-Y. Iu, E. Cecchet, and W. Zwaenepoel. JReq: Database queries in imperative languages. In *CC '10: Proceedings of the 19th International Conference on Compiler Construction*, Berlin, Heidelberg, 2010. Springer-Verlag.

[12] K. Kim, K. Jeon, H. Han, S. gyu Kim, H. Jung, and H. Y. Yeom. MRBench: A benchmark for MapReduce framework. *International Conference on Parallel and Distributed Systems*, 0:11–18, 2008.

[13] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an object-oriented DBMS. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 472–482, New York, NY, USA, 1986. ACM Press.

[14] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[15] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 165–178, New York, NY, USA, 2009. ACM.

[16] J. Persyn. Database sharding at Netlog, with MySQL and PHP. `http://www.jurriaanpersyn.com/archives/2009/02/12/database-sharding-at-netlog-with-mysql-and-php/`.

[17] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.*, 13 (4):277–298, 2005.

[18] Spock Proxy. Spock proxy—a proxy for MySQL horizontal partitioning. `http://spockproxy.sourceforge.net/`.

[19] ST Global. Spider storage engine. `http://spiderformysql.com/`.

[20] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel DBMSs: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[21] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a Map-Reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, 2009.

[22] Transaction Processing Performance Council (TPC). *TPC Benchmark H (Decision Support) Standard Specification Version 2.8.0*. Transaction Processing Performance Council, 2008.

[23] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[24] B. Wiedermann and W. R. Cook. Extracting queries by static analysis of transparent persistence. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 199–210, New York, NY, USA, 2007. ACM Press.

[25] B. Wiedermann, A. Ibrahim, and W. R. Cook. Interprocedural query extraction for transparent persistence. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 19–36, New York, NY, USA, 2008. ACM.

[26] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In R. Draves and R. van Renesse, editors, *OSDI*, pages 1–14. USENIX Association, 2008.