Building a Hybrid Warehouse: Efficient Joins between Data Stored in HDFS and Enterprise Warehouse

YUANYUAN TIAN, IBM Research – Almaden, USA
FATMA ÖZCAN, IBM Research – Alamden, USA
TAO ZOU, Google, USA
ROMULO GONCALVES, The Netherlands eScience Center, Netherlands
HAMID PIRAHESH, IBM Research – Almaden, USA

The Hadoop Distributed File System (HDFS) has become an important data repository in the enterprise as the center for all business analytics, from SQL queries and machine learning to reporting. At the same time, enterprise data warehouses (EDWs) continue to support critical business analytics. This has created the need for a new generation of a special federation between Hadoop-like big data platforms and EDWs, which we call the *hybrid warehouse*. There are many applications that require correlating data stored in HDFS with EDW data, such as the analysis that associates click logs stored in HDFS with the sales data stored in the database. All existing solutions reach out to HDFS and read the data into the EDW to perform the joins, assuming that the Hadoop side does not have efficient SQL support.

In this article, we show that it is actually better to do most data processing on the HDFS side, provided that we can leverage a sophisticated execution engine for joins on the Hadoop side. We identify the best hybrid warehouse architecture by studying various algorithms to join database and HDFS tables. We utilize Bloom filters to minimize the data movement and exploit the massive parallelism in both systems to the fullest extent possible. We describe a new *zigzag join* algorithm and show that it is a robust join algorithm for hybrid warehouses that performs well in almost all cases. We further develop a sophisticated cost model for the various join algorithms and show that it can facilitate query optimization in the hybrid warehouse to correctly choose the right algorithm under different predicate and join selectivities.

CCS Concepts: • **Information systems → Join algorithms**; **MapReduce-based systems**; **Relational parallel and distributed DBMSs**; **Federated databases**; *Query optimization*; *Online analytical processing engines*; *Data warehouses*; DBMS engine architectures;

Additional Key Words and Phrases: Distributed join, join on Hadoop, Bloom filter, SQL-on-Hadoop, hybrid warehouse, federation, query push-down, cost model

ACM Reference Format:
Yuanyuan Tian, Fatma Özcan, Tao Zou, Romulo Goncalves, and Hamid Pirahesh. 2016. Building a hybrid warehouse: Efficient joins between data stored in HDFS and enterprise warehouse. ACM Trans. Database Syst. 41, 4, Article 21 (November 2016), 38 pages.
DOI: http://dx.doi.org/10.1145/2972950

The work described in this article was conducted while Tao Zou and Romulo Goncalves were working at IBM Research – Almaden.
Authors' addresses: Y. Tian, F. Özcan, and H. Pirahesh, IBM Research – Almaden, 650 Harry Road, San Jose, CA 95120; emails: {ytian, fozcan, pirahesh}@us.ibm.com; T. Zou, Google, 1600 Amphitheatre Parkway, Mountain View, CA 94043; email: taozou@google.com; R. Goncalves, Netherlands eScience Center, Science Park 140 (Matrix I), 1098 XG Amsterdam, The Netherlands; email: r.goncalves@esciencecenter.nl.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2016 ACM 0362-5915/2016/11-ART21 $15.00
DOI: http://dx.doi.org/10.1145/2972950

1. INTRODUCTION

The Hadoop Distributed File System (HDFS) has become the *core storage system* for enterprise data, including enterprise application data, social media data, log data, click stream data, and other Internet data. The main reason for the popularity of HDFS is its many merits, including scalability, resiliency, and extremely low cost (commodity hardware and open-source software). Enterprises are using various big data technologies to process data and drive actionable insights. HDFS serves as the storage where other distributed processing frameworks, such as MapReduce [Dean and Ghemawat 2008] and Spark [Zaharia et al. 2012], access and operate on large volumes of data.

At the same time, enterprise data warehouses (EDWs) continue to support critical business analytics. EDWs are usually shared-nothing parallel databases that support complex SQL processing, updates, and transactions (we use EDW and database interchangeably in the remainder of the article). As a result, they manage up-to-date data and support various business analytics tools, such as reporting and dashboards.

Many new applications have emerged, requiring access and correlation of data stored in HDFS and EDWs. For example, a company running an ad campaign may want to evaluate the effectiveness of its campaign by correlating click stream data stored in HDFS with actual sales data stored in the database. These applications, together with the coexistence of HDFS and EDWs, have created the need for a new generation of a special federation between Hadoop-like big data platforms and EDWs, which we call the *hybrid warehouse*.

It is very important to highlight the unique challenges of the hybrid warehouse:

—We are dealing with two completely different systems. A full-pledged relational database (EDW) and an SQL-on-Hadoop query processor have very different characteristics. First, the database owns its data, and hence controls the partitioning and data organization. This is not true for an SQL-on-Hadoop query processor, which works with existing files on HDFS. They do not dictate the physical layout of data, because it is simply prohibitive to convert petabytes of HDFS data to a proprietary format before processing. This is fundamentally different from databases. Second, a database can have indexes, but an SQL-on-Hadoop processor cannot exploit the same record-level indexes. Because they do not control the data, data can be inserted outside the query processor's control. In addition, HDFS does not provide the same level of performance for small reads as a local file system, as it was mainly designed for large scans. Finally, HDFS does not support update in place, and as a result, SQL-on-Hadoop systems are mostly used for processing data that are not or are infrequently updated. These differences between the two systems make this problem both hybrid and asymmetric.

—There are also differences in terms of capacity and cluster setup. While EDWs are typically deployed on high-end hardware, Hadoop/HDFSs are deployed on commodity servers. A Hadoop/HDFS cluster typically has a much larger scale (up to 10,000s of machines), and hence has more storage and computational capacity than an EDW. In fact, today more and more enterprise investment has shifted from EDWs to big data systems like Hadoop.

—Finally, the hybrid warehouse is also not a typical federation between two databases. Existing federation solutions [Josifovski et al. 2002; Adali et al. 1996; Shan et al. 1995; Tomasic et al. 1998; Papakonstantinou et al. 1995] use a client-server model to access the remote databases and move the data. In particular, they use JDBC/ODBC interfaces for pushing down a maximal subquery and retrieving its results. Such a solution ingests the result data serially through the single JDBC/ODBC connection, and hence is only feasible for small amounts of data. In the hybrid warehouse case, a new solution that connects at a lower system layer is needed to exploit the massive

parallelism on both the Hadoop/HDFS side and the EDW side and move data between the two in parallel. This requires nontrivial coordination and communication between the two systems.

1.1. Existing Database and Hadoop Hybrid Solutions

Many existing solutions to integrate HDFS and database data, such as Sqoop[1] and the Teradata connector for Hadoop [Teradata 2013], use utilities to replicate the database data onto HDFS. However, it is not always desirable to empty the warehouse and use HDFS instead, due to the many existing applications that are already tightly coupled to the warehouse. More importantly, HDFS still does not have a good solution to do update in place, whereas warehouses always have up-to-date data.

Conversely, other solutions load HDFS data into the database to perform joins, either in a static fashion prior to querying [Shrinivas et al. 2013; Oracle 2012] or dynamically at query time [Frazier 2013; DeWitt et al. 2013; McClary 2014; Özcan et al. 2011]. These solutions implicitly assume that SQL-on-Hadoop systems do not perform joins efficiently. Although this was true for the early SQL-on-Hadoop solutions, such as Hive [Thusoo et al. 2009], it is not clear whether the same still holds for the current-generation solutions such as IBM Big SQL [Gray et al. 2015], Impala [Kornacker et al. 2015], and Presto [Traverso 2013]. There has been a significant shift during the last two years in the SQL-on-Hadoop solution space, in which these new systems have moved away from MapReduce to have shared-nothing parallel database architectures. They run SQL queries using their own long-running daemons executing on every HDFS DataNode. Instead of materializing intermediate results, these systems pipeline them between computation stages. Moreover, HDFS tables are usually much bigger than database tables, so it is not always feasible to ingest HDFS data and perform joins in the database. Another important observation is that enterprises are investing more on big data systems such as Hadoop and less on expensive EDW systems. As a result, there is more capacity on the Hadoop side. Remotely reading HDFS data into the database introduces significant overhead and burden on the EDWs, because they are already fully utilized by existing applications, and hence carefully monitored and managed.

Note that before the new-generation SQL-on-Hadoop systems, the benefit of applying parallel database techniques to big data processing was already demonstrated by alternative big data platforms to MapReduce, such as Stratosphere [Alexandrov et al. 2014], Asterix [Alsubaiee et al. 2014], and SCOPE [Chaiken et al. 2008].

Splitting query processing between the database and Hadoop has been exploited by PolyBase [DeWitt et al. 2013] to utilize vast Hadoop resources. However, PolyBase only considers pushing down limited functionality to Hadoop, such as selections and projections, and considers pushing down joins only when both tables are stored in HDFS.

1.2. Joins in Hybrid Warehouse

In this article, we envision an architecture of the hybrid warehouse by studying the important problem of efficiently executing joins between HDFS and EDW data.[2] We consider executing the join both in the database and on the HDFS side.

We start by adapting well-known distributed join algorithms and propose extensions that work well in the hybrid warehouse setting. Note that the tradeoffs in design choices for these join algorithms are quite different from the traditional parallel database join algorithms, because we are dealing with two asymmetric, heterogeneous, and independently managed distributed systems in the hybrid warehouse.

[1] http://sqoop.apache.org.

[2] A preliminary version of this article was published in Tian et al. [2015].

Parallel databases use various techniques to optimize joins and minimize data movement. They use broadcast joins when one of the tables participating in the join is small enough, and the other is very large, to save communication cost. The databases also exploit careful physical data organization for joins. They rely on query workloads to identify joins between large tables and copartition them on the join key to avoid data communication at query time.

In the hybrid warehouse, these techniques have limited applicability. Broadcast joins can only be used in limited cases, because the data involved is usually very large. As the database and the HDFS are two independent systems that are managed separately, copartitioning related tables is simply not an option. As a result, we need to adapt existing techniques to optimize the joins between very large tables when neither is partitioned on the join key. It is also very important to note that no EDW in the market today has a good solution for joining two large tables when they are not copartitioned.

We exploit Bloom filters to reduce the data communication costs in joins for hybrid warehouses. A Bloom filter is a compact data structure that allows testing whether a given value is in a set very efficiently, with a controlled false-positive rate. Bloom filters have been proposed in the distributed relational query setting [Mackert and Lohman 1986]. But they are not used widely, because they introduce overhead of extra computation and communication. In this article, we show that Bloom filters are almost always beneficial when communicating data in the hybrid warehouse that integrates two heterogeneous and massively parallel data platforms, as opposed to homogeneous parallel databases. Furthermore, we describe a new join algorithm, the *zigzag join*,[3] which uses Bloom filters both ways to ensure that only the records that will participate in the join need to be transferred through the network. The zigzag join is most effective when the tables involved in the join do not have good local predicates to reduce their sizes but the join itself is selective.

We implemented the proposed join algorithms for the hybrid warehouse using a commercial shared-nothing parallel database, IBM DB2 LUW, as the EDW. To expedite the prototyping efforts, and to enable portability of our solutions to multiple databases, we implemented the database-side operations of the joins using the extensibility provided by user-defined functions (UDFs). Note that most EDWs today have UDF support for extensibility. On the HDFS/Hadoop side, we took a prototype of the I/O layer and the scheduler from an existing SQL-on-Hadoop system, IBM Big SQL [Gray et al. 2015], and extended it with our own runtime for executing joins, which is able to pipeline operations and overlay network communication with processing and data scanning. Moreover, to enable joins across the two systems, we also augmented the HDFS-side execution engine with a coordination service, as well as the functionality to move data efficiently to and from databases in a parallel streaming fashion. This augmented engine is called JEN. We observe that with such a sophisticated execution engine on HDFS, it is actually often better to execute the joins on the Hadoop side. This is a finding that challenges the common wisdom from existing database and Hadoop hybrid solutions.

Finally, to choose the most appropriate algorithm in different query settings, we further developed a sophisticated cost model for the proposed join algorithms and verified its effectiveness through experiments. This cost model goes way beyond that of a parallel database for joins due to the complexity introduced by the hybrid warehouse architecture. It needs to capture the resource consumptions not only within each of the two distributed systems but also across the two. In addition, it needs to take the individual capabilities and the asymmetry between the two systems into account.

[3]Zigzag join is a research name for this particular proposed algorithm and needs to be distinguished from the zigzag join in DB2 LUW (http://www.ibm.com/developerworks/data/library/techarticle/dm-1303zigzag).

Moreover, we also design the cost model to fit in a standard optimizer framework, so that it can be easily plugged into a modular or standalone query optimizer designed for big data, such as Orca [Soliman et al. 2014]. This cost model constitutes an important and necessary building block for a future optimizer in the hybrid warehouse architecture.

1.3. Contributions

The contributions of this article are summarized as follows:

—We revisit the join algorithms that are used in distributed query processing and adapt them to work in the hybrid warehouse between two heterogeneous massively parallel data platforms. We utilize Bloom filters to minimize the data movement, and exploit the massive parallelism in both systems.
—We describe a new join algorithm, the zigzag join, which uses Bloom filters on both sides, and provide a very efficient implementation that minimizes the overhead of Bloom filter computation and exchange. We show that the zigzag join algorithm is a robust algorithm that performs well for hybrid warehouses in most cases.
—Through detailed experiments, we show that it is often better to execute joins on the HDFS/Hadoop side as data size grows, provided that there is a sophisticated execution engine on the HDFS side. To the best of our knowledge, this is the first work in the hybrid warehouse that systematically studies join algorithms and argues for a solution that divides the computation between the two systems.
—We propose an architecture for the hybrid warehouse where EDW and Hadoop clusters jointly execute the joins. We describe the system called JEN, in which we implement all the proposed join algorithms. We want to highlight that JEN is more than a sophisticated HDFS-side execution engine for joins, which exploits the various optimization strategies employed by a shared-nothing parallel database architecture, including multithreading, pipelining, hash-based aggregation, and so forth. JEN also contains two crucial components that enable the coordination and communication between the two independently managed distributed systems to support joins across. We envision that existing SQL-on-Hadoop systems can be augmented with the capabilities of JEN.
—We develop a sophisticated cost model for the various join algorithms by characterizing the resource consumptions within each distributed system, as well as between the two. Through experiments, we demonstrate that the cost model is able to capture the relative performance of the join algorithms and correctly pick the best algorithm under different predicate and join selectivity scenarios. As far as we know, this work is the first to develop a cost model for join algorithms in the hybrid warehouse setting.

The rest of the article is organized as follows: We start with a concrete example scenario, including our assumptions, in Section 2. The join algorithms are discussed in Section 3. We implemented our algorithms using a commercial parallel database and our own join execution engine on HDFS. In Section 4, we describe this implementation. The cost model of proposed join algorithms is presented in Section 5. We provide detailed experimental results in Section 6 and discuss the insights learned in Section 7. Related work is described in Section 8. Finally, we conclude in Section 9.

2. AN EXAMPLE SCENARIO

In this article, we study the problem of joins in the hybrid warehouse. We will use the following example scenario to illustrate the kind of query workload we focus on. This example represents a wide range of real application needs.

Consider a retailer, such as Walmart or Target, that sells products in local stores as well as online. All the transactions, either offline or online, are managed and stored in a parallel database, whereas users' online click logs are captured and stored in HDFS.

The retailer wants to analyze the correlation of customers' online behaviors with sales data. This requires joining the transaction table T in the parallel database with the log table L on HDFS. One such analysis can be expressed as the following SQL query:

```
SELECT L.urlPrefix, COUNT(*)
FROM T, L
WHERE T.category = 'Canon Camera'
AND region(L.ip) = 'East Coast'
AND T.uid = L.uid
AND T.tDate >= L.lDate AND T.tDate <= L.lDate + 1
GROUP BY L.urlPrefix
```

This query tries to find out the number of views of the URLs visited by customers with IP addresses from the East Coast who bought Canon cameras within 1 day of their online visits.

Now, we look at the structure of the example query. It has local predicates on both tables, followed by an equi-join. The join is also coupled with predicates on the joined result, as well as group-by and aggregation. In this article, we will describe our algorithms using this example query.

In common setups, a parallel database is deployed on a small number (10s to 100s) of high-end servers, whereas HDFS resides on a large number (100s to 10,000s) of commodity machines. We assume that the parallel database is a full-fledged shared-nothing parallel database. It has an optimizer, indexing support, and sophisticated SQL engine. HDFS, on the other hand, is optimized for large bulk I/O, and as a result, record-level indexing does not provide significant performance benefits. Therefore, we assume a scan-based processing engine on HDFS. In fact, this is the case for all the existing SQL-on-Hadoop systems, such as MapReduce-based Hive [Thusoo et al. 2009], Spark SQL [Armbrust et al. 2015], and Impala. We do not tie the join algorithm descriptions to a particular processing framework; thus, we generalize any scan-based distributed data processor on HDFS as an HQP (HDFS Query Processor).

For data characteristics, we assume that both tables are large, but the HDFS table is much larger, which is the case in most realistic scenarios. In addition, since we focus on analytic workloads, we assume there is always group-by and aggregation at the end of the query. As a result, the final query result is relatively small. Finally, without loss of generality, we assume that queries are issued at the parallel database side and the final results are also returned at the database side. Note that forwarding a query from the database to HDFS is relatively cheap; so is passing the final results from HDFS back to the database.

Note that in this article we focus on the join algorithms for hybrid warehouses; thus, we only include a two-way join in the example scenario. Real big-data queries may involve joining multiple tables. For these cases, we need to rely on the query optimizer in the database to decide on the right join orders, since queries are issued at the database side in our setting. However, the study of the join orders in a hybrid warehouse is beyond the scope of this article.

3. JOIN ALGORITHMS

In this section, we describe a number of algorithms for joining a table stored in a shared-nothing parallel database with another table stored in HDFS. We start by adapting well-known distributed join algorithms and explore ways to minimize the data movement between these two systems by utilizing Bloom filters. While existing approaches [Mullin 1990; Michael et al. 2007; Mackert and Lohman 1986; Li and Ross 1995; Polychroniou et al. 2014] were designed for homogeneous environments, our join algorithms work across two heterogeneous systems in the hybrid warehouse. The

Table I. Notations Used in the Description of Algorithms

T	the database table T
L	the HDFS table L
$pred_T$	local predicates on T
$pred_L$	local predicates on L
$proj_T$	local projection on T
$proj_L$	local projection on L
T'	the database table after applying local predicates $pred_T$ and local projection $proj_T$ on T
L'	the HDFS table after applying local predicates $pred_L$ and local projection $proj_L$ on L
$bf_{T'}$	the Bloom filter computed on T'
$bf_{L'}$	the Bloom filter computed on L'
T''	the resulting database table by applying Bloom filter $bf_{L'}$ on T'
L''	the resulting HDFS table by applying Bloom filter $bf_{T'}$ on L'

design of these algorithms seeks to leverage the processing power of both systems and maximize parallel execution.

Before we describe the join algorithms, let's provide a brief introduction to Bloom filters first. A Bloom filter [Bloom 1970] is essentially a bit array of x bits with k hash functions defined to summarize a set of elements. Adding an element to the Bloom filter involves applying the k hash functions on the element and setting the corresponding positions of the bit array to 1. Symmetrically, testing whether an element belongs to the set requires simply applying the hash functions and checking whether all of the corresponding bit positions are set to 1. Obviously, the testing incurs some false positives. However, the false-positive rate can be computed based on x, k, and y, where y is the number of unique elements in the set. Therefore, x and k can be tuned for the desired false-positive rate. This technology is very effective for highly selective joins. By building a Bloom filter on the join keys of one table, we can use it to prune out the nonjoinable records from the other table.

In the description of the algorithms later, we use T to denote the database table, and L to stand for the HDFS table. We represent the local predicates on T as $pred_T$, and the projections on T as $proj_T$. The database table after applying local predicates $pred_T$ and local projection $proj_T$ on T is denoted as T'. We define $pred_L$, $proj_L$, and L' similarly as earlier. We use $bf_{T'}$ and $bf_{L'}$ to symbolize the Bloom filters computed on T' and L', respectively. Finally, we denote the resulting database table by applying Bloom filter $bf_{L'}$ on T' as T'', and the resulting HDFS table by applying Bloom filter $bf_{T'}$ on L' as L''. These notations are summarized in Table I.

3.1. DB-Side Join

Many database/HDFS hybrid systems, including Microsoft PolyBase [DeWitt et al. 2013], Pivotal HAWQ,[4] Teradata SQL-H [Frazier 2013], and Oracle Big Data SQL [McClary 2014], fetch the HDFS table and execute the join in the database. We first explore this approach, which we call DB-side join. In the plain version, the HDFS side applies local predicates and projection and sends the filtered HDFS table in parallel to the database, where the join is carried out using the algorithm chosen by the database query optimizer. The performance of this join method is dependent on the amount of data that needs to be transferred from HDFS. Two factors determine this size: the selectivity of the local predicates $pred_L$ over the HDFS table L and the size of the projected columns.

Note that the HDFS table L is usually much larger than the database table T. Even if the local predicates $pred_L$ are highly selective, the filtered HDFS table L' can still be

[4]http://pivotal.io/big-data/white-paper/a-true-sql-engine-for-hadoop-pivotal-hd-hawq.

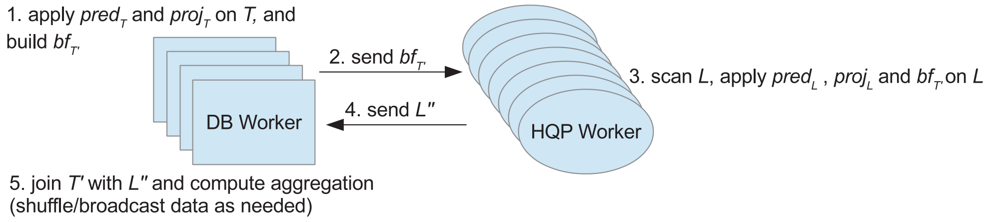


Fig. 1. Data flow of DB-side join with Bloom filter.

quite large. In order to further reduce the amount of data transferred from HDFS to the parallel database, we introduce a Bloom filter $bf_{T'}$ on the join key of T', which is the database table after applying local predicates and projection, and send the Bloom filter to the HDFS side. This technique filters out HDFS records that cannot be joined. This DB-side join algorithm is illustrated in Figure 1.

In this DB-side join algorithm, parallel database nodes (DB workers in Figure 1) first compute the Bloom filter for their local partitions and then aggregate them into a global Bloom filter ($bf_{T'}$) by simply applying bitwise OR. This Bloom filter is used to further trim down the size of the HDFS data into L'', in addition to the local predicates $pred_L$ and projection $proj_L$. After the filtered HDFS data L'' is brought into the database, it is joined with the database data T' using the join algorithm chosen by the query optimizer. For example, when one table is much smaller, the optimizer may choose to broadcast the smaller table to all database workers, then perform the join locally on each worker (broadcast join). Otherwise, both tables can be repartitioned and redistributed across the workers, before the local joins are executed (repartition join). Note that in the DB-side join, the HDFS data may need to be shuffled again at the database side before the join (e.g., if repartition join is chosen by the optimizer), because HQP nodes do not have access to the partitioning hash function of the database. Even if the hash function were exposed, the database would not take advantage of it for optimization, as the optimizer doesn't know how the data is partitioned coming from the remote source.

In the previous algorithm, there are different ways to send the database Bloom filter to HDFS and transmit the HDFS data to the database. Which approach works best depends on the network topology and the bandwidth. We defer the discussion of detailed implementation choices to Section 4.

3.2. HDFS-Side Broadcast Join

The second algorithm is called HDFS-side broadcast join, or simply broadcast join. It executes the join on the HDFS side. The rationale behind this algorithm is that if the predicates $pred_T$ on the database table T are highly selective, the filtered database data T' is small enough to be sent to every HQP node, so that only local joins are needed without any shuffling of the HDFS data. When the join is executed on the HDFS side, it is logical to push down the grouping and aggregation to the HDFS side as well. This way, only a small amount of summary data needs to be transferred back to the database to be returned to the user. The HDFS-side broadcast join algorithm is illustrated in Figure 2.

In the first step, each database node applies local predicates $pred_T$ and projection $proj_T$ over its partition of the database table T. Each database node then broadcasts its filtered partition to every HQP node (Step 2). In Step 3, each HQP node performs a local join. Group-by and partial aggregation are also carried out on the local data in this step. The final aggregation is computed in Step 4 and sent to the database in Step 5.

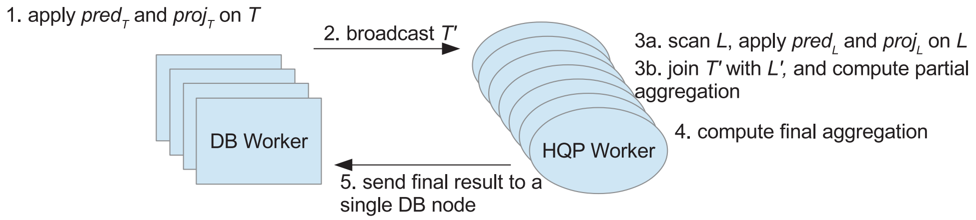


Fig. 2. Data flow of HDFS-side broadcast join.

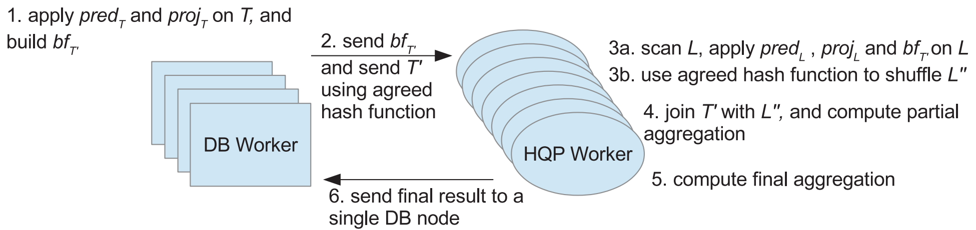


Fig. 3. Data flow of HDFS-side repartition join with Bloom filter.

3.3. HDFS-Side Repartition Join

The second HDFS-side algorithm we consider is the HDFS-side repartition join, or simply repartition join. If the local predicates $pred_T$ over the database table T are not highly selective, then broadcasting the filtered data T' to all HQP nodes is not a good strategy. In this case, we need a robust join algorithm. We expect the HDFS table L to be much larger than the database table T in practice, and hence it makes more sense to transfer the smaller database table to HDFS and execute the join on the HDFS side. Just as in the DB-side join, we can also improve this basic version of repartition join by introducing a Bloom filter. Figure 3 demonstrates this improved algorithm.

In Step 1, all database nodes apply local predicates over the database table T and project out the required columns, resulting in T'. All database nodes also compute their local Bloom filters, which are then aggregated into a global Bloom filter $bf_{T'}$ and sent to the HQP nodes. In this algorithm, the HDFS side and the database agree on the hash function to use when shuffling the data. In Step 2, all database nodes use this agreed-upon hash function and send their data to the identified HQP nodes. This means that once the database data reaches the HDFS side, it doesn't need to be reshuffled among the HQP nodes. In Step 3 of the HDFS-side repartition join, all HQP nodes apply the local predicates and projections over the HDFS table as well as the Bloom filter $bf_{T'}$ sent by the database. The Bloom filter further filters out the HDFS data into L''. The HQP nodes use the same hash function to shuffle the filtered HDFS table L''. Then, they perform the join and partial aggregation (Step 4). The final aggregation is executed on the HDFS side in Step 5 and sent to the database in Step 6.

3.4. HDFS-Side Zigzag Join

When local predicates on neither the HDFS table nor the database table are selective, we need to fully exploit the join selectivity to perform the join efficiently. In some sense, a selective join can be used as if it were extra local predicates on both tables. To illustrate this point, let's first introduce the concepts of *join-key selectivity* and *join-key predicate*.

We define $JK(T')$ as the set of join keys in T', which is the table after applying local predicates and projection on the database table T. Similarly, $JK(L')$ is defined

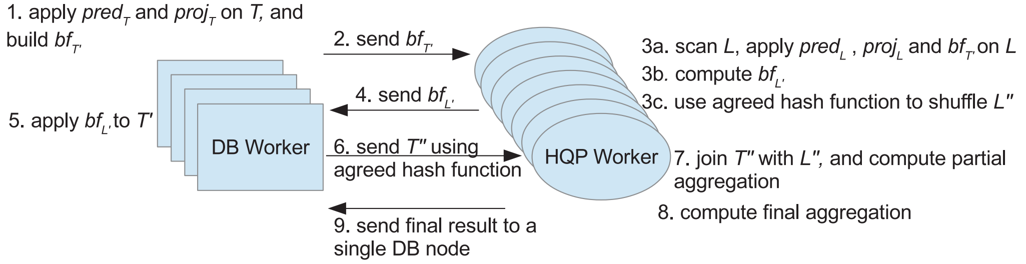


Fig. 4. Data flow of zigzag join.

as the set of join keys in L'. We know that only the join keys in $JK(T') \cap JK(L')$ will appear in the final join result. So, only the $(JK(T') \cap JK(L'))/JK(L')$ fraction of the unique join keys in L' will participate in the join. We call this fraction the *join-key selectivity* on L', denoted as $S_{L'}$. Likewise, the join-key selectivity on T' is $S_{T'} = (JK(T') \cap JK(L'))/JK(T')$. Leveraging the join-key selectivities through Bloom filters is essentially like applying extended local predicates on the join key columns of both tables. We call them *join-key predicates*.

Through the use of a one-way Bloom filter, the DB-side join and the repartition join described in previous sections are only able to leverage the HDFS-side join-key predicate to reduce either the HDFS data transferred to the database or the HDFS data shuffled among the HQP workers. The DB-side join-key predicate is not utilized at all. Later, we introduce a new algorithm, zigzag join, to fully utilize the join-key predicates on both sides in reducing data movement, through the use of two-way Bloom filters. Again, we expect the HDFS table to be much larger than the database table in practice, and hence the join in this algorithm is executed on the HDFS side, and both sides agree on the hash function to send data to the correct HQP nodes for the final join.

The zigzag join algorithm is described in Figure 4. In Step 1, all database nodes apply local predicates and projection, and compute their local Bloom filters. The database then computes the global Bloom filter $bf_{T'}$ and sends it to all HQP nodes in Step 2. As in the repartition join with Bloom filter, this Bloom filter helps reduce the amount of HDFS data that needs to be shuffled.

In Step 3, all HQP nodes apply their local predicates, projection, and the database Bloom filter $bf_{T'}$ over the HDFS table and compute a local Bloom filter for the HDFS table. The local Bloom filters are aggregated into a global one, $bf_{L'}$, which is sent to all database nodes. At the same time, the HQP nodes shuffle the filtered HDFS table L'' using the agreed-upon hash function. In Step 5, the database nodes receive the HDFS Bloom filter $bf_{L'}$ and apply it to the database table T' to further reduce the number of database records that need to be sent. The application of Bloom filters on both sides ensures that only the data that will participate in the join (subject to false positives of the Bloom filter) needs to be transferred.

Note that in Step 5 the database data needs to be accessed again. We rely on the advanced database optimizer to choose the best strategy: either to materialize the intermediate table T' after local predicates and projection are applied or to utilize indexes to access the original table T. It is also important to note that while the HDFS Bloom filter is applied to the database data, the HQP nodes can be shuffling the HDFS data in parallel, hence overlapping many steps of the execution.

In Step 6, the database nodes send the further filtered database data T'' to the HQP nodes using the agreed-upon hash function. The HQP nodes perform the join and partial aggregation (Step 7), collaboratively compute the global aggregation (Step 8), and finally send the result to the database (Step 9).

Note that zigzag join is the only join algorithm that can fully utilize the join-key predicates as well as the local predicates on both sides to reduce network traffic. The HDFS data shuffled across HQP nodes are filtered by the local predicates $pred_L$ on L, the local predicates $pred_T$ on T (as $bf_{T'}$ is built on T'), and the join-key predicate on L'. Similarly, the database records transferred to the HDFS side are filtered by the local predicates $pred_T$ on T, the local predicates $pred_L$ on L (as $bf_{L'}$ is built on L'), and the join-key predicate on T'.

Although Bloom filters and semijoin techniques are known in the literature, they are not widely used in practice due to the overhead of computing Bloom filters and multiple data scans. However, the asymmetry of slow HDFS table scan and fast database table access makes these techniques more desirable in a hybrid warehouse. Note that a variant version of the zigzag join algorithm that executes the final join on the database side will not perform well, because scanning the HDFS table twice, without the help of indexes, is expected to introduce significant overhead.

4. IMPLEMENTATION

In this section, we provide an overview of our implementation of the join algorithms for the hybrid warehouse and highlight some important details.

4.1. Overview

In our implementation, we used IBM DB2 LUW with the Database Partitioning Feature (DPF) [Baru et al. 1995], which is a shared-nothing distributed version of DB2, as our EDW. To expedite the prototyping efforts and ensure the portability of our solution to other EDWs, we leverage user-defined functions (UDFs) supported in most EDWs for implementing database-side operations of joins. In fact, we implemented all the aforementioned join algorithms using C UDFs in DB2 DPF and our own C++ MPI-based join execution engine on HDFS, called JEN. In addition to being our specialized implementation of HQP used in the algorithm descriptions in Section 3, JEN also has the crucial capabilities to coordinate with the EDW and move data efficiently between the two systems. We used a prototype of the I/O layer and the scheduler from an early version of IBM Big SQL 3.0 [Gray et al. 2015] and built JEN on top of them. We also utilized Apache HCatalog[5] to store the metadata of the HDFS tables.

JEN consists of a single coordinator and a number of workers, with each worker running on an HDFS DataNode. JEN workers are responsible for reading parts of HDFS files, executing local query plans, and communicating with other workers, the coordinator, and DB2 DPF workers. Each JEN worker is multithreaded, capable of exploiting all the cores on a machine. The communication between two JEN workers or with the coordinator is done through TCP/IP sockets. The JEN coordinator has multiple roles. First, it is responsible for managing the JEN workers and their states so that workers know which other workers are up and running in the system. Second, it serves as the central contact for the JEN workers to learn the IPs of the DB2 workers and vice versa, so that they can establish communication channels for data transfers. Third, it is also responsible for retrieving the metadata (HDFS path, input format, etc.) for HDFS tables from HCatalog. Once the coordinator knows the path of the HDFS table, it contacts the HDFS NameNode to get the locations of each HDFS block and evenly assigns the HDFS blocks to the JEN workers to read, respecting data locality.

On the DB2 side, we leverage the existing database query engine as much as possible. For the functionalities not provided, such as computing and applying Bloom filters, and different ways of transferring data to and from JEN workers, we implemented them using *unfenced* C UDFs, which provide performance close to built-in functions,

[5]http://cwiki.apache.org/confluence/display/Hive/HCatalog.

as they run in the same process as the DB2 engine. The communication between a DB2 DPF worker and a JEN worker is also through TCP/IP sockets. To exploit the multicores on a machine, we set up multiple DB2 workers on each machine of a DB2 DPF cluster, instead of one DB2 worker enabled with multicore parallelism. This is mainly to simplify our C UDF implementations, as otherwise we have to deal with intraprocess communications inside a UDF.

Each of the join algorithms is invoked by issuing a *single* query to DB2. With the help of UDFs, this single query executes the *entire* join algorithm: initiating the communication between the database and the HDFS side, instructing the two sides to work collaboratively, and finally returning the results back to the user.

4.1.1. The DB-Side Join Example. Let's use an example to illustrate how the database side and the HDFS side collaboratively execute a join algorithm. If we want to execute the example query in Section 2 using the DB-side join with Bloom filter, we submit the following SQL query to DB2:

```
WITH LocalFilter(lf) AS (
  SELECT get_filter(MAX(calc_filter(uid)))
  FROM T
  WHERE T.category='Canon Camera'
  GROUP BY DBPARTITIONNUM(tid)
),
GlobalFilter(gf) AS (
  SELECT *
  FROM TABLE(SELECT combine_filter(lf) FROM LocalFilter)
  WHERE gf IS NOT NULL
),
Clicks(uid, urlPrefix, lDate) AS (
  SELECT uid, urlPrefix, lDate
  FROM GlobalFilter,
       TABLE(read_hdfs('L', 'region(ip)= \'East Coast\'','uid, urlPrefix,
             lDate', GlobalFilter.gf, 'uid'))
)
SELECT urlPrefix, COUNT(*)
FROM Clicks, T
WHERE T.category='Canon Camera'
      AND Clicks.uid=T.uid
      AND DAYS(T.tDate)-DAYS(Clicks.lDate)>=0
      AND DAYS(T.tDate)-DAYS(Clicks.lDate)<=1
GROUP BY urlPrefix
```

In this SQL query, we assume that the database table T is distributed across multiple DB2 workers on the `tid` field. The first subquery (`LocalFilter`) uses two scalar UDFs *`calc_filter`* and *`get_filter`* together to compute a Bloom filter on the local partition of each DB2 worker. We enabled the two UDFs to execute in parallel, and the statement `GROUP BY DBPARTITIONNUM(tid)` further makes sure that each DB2 worker computes the Bloom filter on its local data in parallel. The second subquery (`GlobalFilter`) uses another scalar UDF *`combine_filter`* to combine the local Bloom filters into a single global Bloom filter, which is returned as a single row in `GlobalFilter`. By declaring *`combine_filter`* "disallow parallel," we make sure it is executed once on one of the DB2 workers (all local Bloom filters are sent to a single DB2 worker). In the third subquery (`Clicks`), a table UDF, *`read_hdfs`*, is used to pass the following information to the HDFS side: the name of the HDFS table, the local predicates on the HDFS

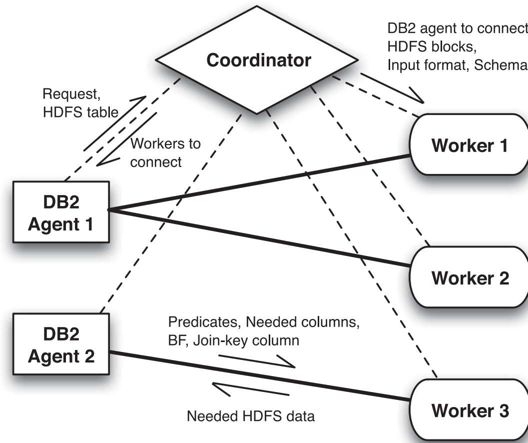


Fig. 5. Communication in the `read_hdfs` UDF of the DB-side join with Bloom filter.

table, the projected columns needed to be returned, the global database Bloom filter, and the join-key column on which the Bloom filter needs to be applied. In the same UDF, the JEN workers subsequently read the HDFS table and send the required data after applying predicates, projection, and the Bloom filter back to the DB2 workers. The *read_hdfs* UDF is executed on each DB2 worker in parallel (the global Bloom filter is broadcast to all DB2 workers) and carries out the parallel data transfer from HDFS to DB2. After that, the join, together with the group-by and aggregation, is executed at the DB2 side. We pass a hint of the cardinality information to the *read_hdfs* UDF (by using the CARDINALITY statement in the UDF definition), so that the DB2 optimizer can choose the right plan for the join. The final result is returned to the user on the database side.

Now let's look into the details of the *read_hdfs* UDF. Since there is only one record in `GlobalFilter`, this UDF is called once per DB2 worker. When it is called on each DB2 worker, it first contacts the JEN coordinator to request the connection information to the JEN workers. In return, the coordinator tells each DB2 worker which JEN worker(s) to connect to and notifies the corresponding JEN workers to prepare for the connections from the DB2 workers. This process is shown in Figure 5. Without loss of generality, let's assume that there are m DB2 workers and n JEN workers, and that $m \leq n$. For the DB-side join, the JEN coordinator evenly divides the n workers into m groups. Each DB2 worker establishes connections to all the workers in one group, as illustrated in Figure 5. After all the connections are established, each DB2 worker multicasts the predicates on the HDFS table, the required columns from the HDFS table, the database Bloom filter, and the join-key column to the corresponding group of JEN workers. At the same time, DB2 workers tell the JEN coordinator which HDFS table to read. The coordinator contacts HCatalog to retrieve the paths of the corresponding HDFS files and the input format, and asks the HDFS NameNode for the storage locations of the HDFS blocks. Then, the coordinator assigns the HDFS blocks and sends the assignment as well as the input format to the workers. After receiving all the necessary information, each JEN worker is ready to scan its share of the HDFS data. As it scans the data, it directly applies the local predicates and the Bloom filter from the database side and sends the records with required columns back to its corresponding DB2 worker.

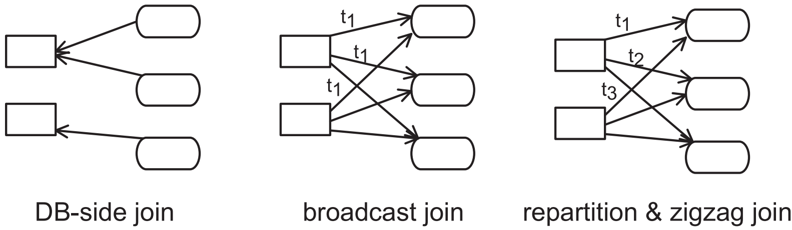


Fig. 6. Data transfer patterns between DB2 workers and JEN workers in the join algorithms.

4.2. Locality-Aware Data Ingestion from HDFS

As our join execution engine on HDFS is scan based, efficient data ingestion from HDFS is crucial for performance. We purposely deploy the JEN workers on all HDFS DataNodes so that we can leverage data locality when reading. In fact, when the JEN coordinator assigns the HDFS blocks to workers, it carefully considers the locations of each HDFS block to create balanced assignments and maximize the locality of data in a best-effort manner. Using this locality-aware data assignment, each JEN worker mostly reads data from local disks. We also enabled short-circuit local reads[6] for HDFS DataNodes to improve the local read speed. To further boost the data ingestion throughput, our data ingestion component uses multiple threads when multiple disks are used for each DataNode.

4.3. Data Transfer Patterns

In this subsection, we discuss the data transfer patterns of different join algorithms. There are three types of data transfers that happen in all the join algorithms: among DB2 workers, among JEN workers, and between DB2 workers and JEN workers. For the data transfers among DB2 workers, we simply rely on DB2 to choose and execute the right transfer mechanisms. Among the JEN workers, there are three places that data transfers are needed: (1) shuffle the HDFS data for the repartition join (with and without Bloom filter) and the zigzag join, (2) aggregate the global HDFS Bloom filter for the zigzag join, and (3) compute the final aggregation result from the partial results on JEN workers in the broadcast join, the repartition join, and the zigzag join. For (1), each worker simply maintains TCP/IP connections to all other workers and shuffles data through these connections. For (2) and (3), each worker sends the local results (either local Bloom filter or local aggregates) to a single designated worker chosen by the coordinator to finish the final aggregation.

The more interesting data transfers happen between DB2 workers and JEN workers. Again, there are three places that the data transfer is needed: shipping the actual data (HDFS or database), sending the Bloom filters, and transmitting the final aggregated results to the database for all the HDFS-side joins. Since Bloom filters and final aggregated results are much smaller than the actual data, transferring them has little impact on the overall performance. For the database Bloom filter sent to HDFS, we multicast the database Bloom filters to HDFS following the mechanism shown in Figure 5. For the HDFS Bloom filter sent to the database, we broadcast the HDFS Bloom filter from the designated JEN worker to all the DB2 workers. The final results on HDFS are simply transmitted from the designated JEN worker to a designated DB2 worker. We focus more on the mechanism for shipping the actual data between DB2 and HDFS. Figure 6 demonstrates the different patterns for transferring the actual data in the different join algorithms.

[6]http://hadoop.apache.org/docs/r2.5.2/hadoop-project-dist/hadoop-hdfs/ShortCircuitLocalReads.html.

DB-side join with and without Bloom filter. For the two DB-side joins, we randomly partition the set of JEN workers into m roughly even groups, where m is the number of DB2 workers, then let each DB2 worker bring in its assigned part of the HDFS data in parallel from the corresponding group of JEN workers. DB2 can choose whatever algorithms for the join that it sees fit, based on data statistics. For example, when the database data is much smaller than HDFS data, the optimizer chooses to broadcast the database table for the join. When the HDFS data is much smaller than the database data, broadcasting the HDFS data is used. In the other cases, a repartition-based join algorithm is chosen. This means that when the HDFS data is transferred to the database side, it may need to be shuffled again among the DB2 workers. To avoid this second data transfer, we would have to expose the partitioning scheme of DB2 to JEN and teach the DB2 optimizer that the data received from JEN workers has already been partitioned in the desired way. Our implementation does not modify the DB2 engine, so we stick with this simpler and noninvasive data transfer scheme for the DB-side joins.

Broadcast join. There are multiple ways to broadcast the database data to JEN workers. One way is to let each DB2 worker connect to all the JEN workers and deliver its data to every worker. Another way is to have each DB2 worker only transfer its data to one JEN worker, which further passes on the data to all other workers. The second approach puts less stress on the interconnection between DB2 and HDFS but introduces a second round of data transfer among the JEN workers. We found empirically that broadcast join only works better than other algorithms when the database table after local predicates and projection is very small. For that case, even the first transfer pattern does not put much strain on the inter-connection between DB2 and HDFS. Furthermore, the second approach actually introduces extra latency because of the extra round of data transfer. For these reasons, we use the first data transfer scheme in our implementation of the broadcast join.

Repartition join with/without Bloom filter and zigzag join. For these three join algorithms, the final join happens at the HDFS side. Before the data transfer starts, the DB2 workers query the JEN coordinator for the hash function used for shuffling data in JEN. When a database record is sent to the HDFS side, the DB2 worker uses the hash function to identify the destination JEN worker.

4.4. Pipelining and Multithreading in JEN

In the implementation of JEN, we try to pipeline operations and parallelize computation as much as possible. Let's take the sophisticated zigzag join as an example.

At the beginning, every JEN worker waits to receive the global Bloom filter from DB2, which is a blocking operation, since all the remaining operations depend on this Bloom filter. After the Bloom filter is obtained, each worker starts to read its portion of the HDFS table (mostly from local disks) immediately. The data ingestion component is able to dedicate one *read thread* per disk when multiple disks are used for an HDFS DataNode. In addition, a separate *process thread* is used to parse the raw data into records, based on the input format and schema of the HDFS table. Then it applies the local predicates, projection, and database Bloom filter on each record. For each projected record that passes all the conditions, this thread uses it to populate the HDFS-side Bloom filter and applies the shuffling hash function on the join key to figure out which JEN worker this record needs to be sent to for the repartition-based join. Then, the record is put in a send buffer ready to be sent. All of these operations on a record are pipelined inside the process thread. At the same time, a pool of *send threads* poll the sending buffers to carry out the data transfers. Another pool of *receive threads* simultaneously receives records from other workers. And for each record received, the receive thread uses the record to build hash tables for the join. The multithreading in this stage of the zigzag join is illustrated in Figure 7. As can be seen, scanning,

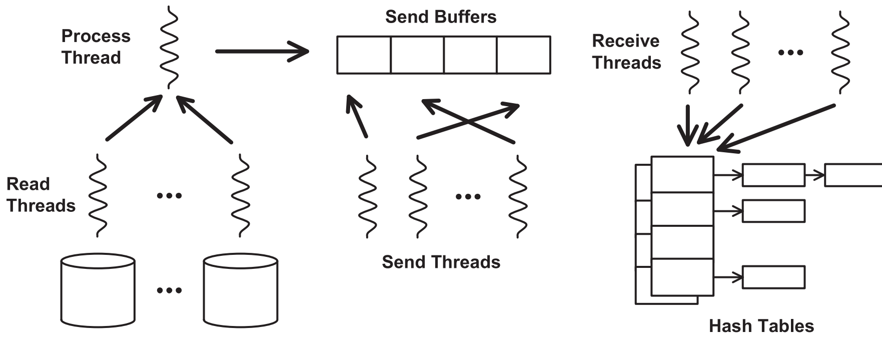


Fig. 7. Interleaving of scanning, processing, and shuffling of HDFS data in zigzag join.

processing, and shuffling (sending and receiving) of HDFS data are carried out totally in parallel. In fact, the repartition join (with/without Bloom filter) also shares the similar interleaving of scanning, processing, and shuffling of HDFS data. Note that reading from HDFS and shuffling data through networks are expensive operations; although we only have one process thread that applies the local predicates, Bloom filter, and projection, it is never the bottleneck.

As soon as the reading from HDFS finishes (read threads are done), a local Bloom filter is built on each worker. The workers send local Bloom filters to a designated worker to compute the global Bloom filter and pass it on to the DB2 workers. After that, every worker waits to receive and buffer the data from DB2 in the background. Once the local hash tables are built (the send and receive threads in Figure 7 are all done), the received database records are used to probe the hash tables, produce join results, and subsequently apply a hash-based group-by and aggregation immediately. Here again, all the operations on a database record are pipelined. When all the local aggregates are computed, each worker sends its partial result to a designated worker, which computes the final aggregate and sends to a single DB2 worker to return to the user.

4.5. Local Join on Each JEN Worker

For all the HDFS-side join algorithms, the final step of the join is carried out locally on each JEN worker. We use an algorithm similar to the Hybrid hash join proposed in DeWitt et al. [1984]. This algorithm does not require all data to fit in memory on each worker. We use a second hash function (different from the hash function for joins) to first copartition data from both tables, then perform a hash join per partition. When a tuple from the *build* side arrives, we first use this hash function to partition it and then put it into the hash table of the corresponding partition. When there is not enough memory to hold all the hash tables, we spill some partitions to local disk. Later, when the tuples from the *probe* side come, we use the same hash function to partition them. If a tuple is hashed to a partition that has its build-side hash table in memory, it is joined directly to the tuples in the hash table. Otherwise, it is written to its partition on local disk. After all probe-side tuples arrive and are joined with the in-memory part of the build-side table, we read in pairs of on-disk partitions of both tables and join them.

For the broadcast join and the two versions of repartition join, we use the filtered database table as the *build* table for the final local join, as the receiving of the database tuples starts before the shuffling of the HDFS data. On the other hand, in the zigzag join, we choose to build the hash tables from the filtered HDFS data and use the transferred database data to probe the hash table, although the database data is

expected to be smaller in most cases. This is because the filtered HDFS data is already being received during the scan of the HDFS table, due to multithreading. Empirically, we find that the receiving of the HDFS data is usually done soon after the scan is finished. On the other hand, the database data will not start to arrive until the HDFS table scan is done, as the HDFS-side Bloom filter is fully constructed only after all HDFS data are processed. Therefore, it makes more sense to start building the hash tables on the filtered HDFS data while waiting for the later arrival of the database data.

5. COST MODEL

In this section, we present a cost model for the different join algorithms we proposed for hybrid warehouses. As in previous work on cost models in distributed settings [Mackert and Lohman 1986], we estimate the *total resource time* (total sum of time consumed by various resources, including disk I/O, network I/O, and CPU), in milliseconds, of each join algorithm. In our cost formulas, total resource time does not capture the overlap of consuming different resources, such as the interleaving of CPU and I/O in a multithreaded setting, because the amount of such overlap is nondeterministic and hard to model very accurately.

In addition to the notations in Table I, we also introduce the following notations for the cost model. We use σ_T to represent the selectivity of the predicates $pred_T$ on the database table T, and π_T to denote the reduction factor of the projection $proj_T$ on T. We define $|T|$ as the cardinality of T, and $\|T\|$ as the size of T in bytes on disk in its storage format. Similar definitions apply to the HDFS table L. We use n to denote the number of workers on the HDFS side and m to represent the number of database workers.

With this notation, we can estimate the size of intermediate results. The database table after projection and predicates, denoted as T', has $|T'| = |T| \times \sigma_T$ records and is of size $\|T'\| = \|T\| \times \sigma_T \times \pi_T \times \lambda_T$, where λ_T is the ratio of the uncompressed size of T to the on-disk size of T (e.g., when T is compressed on disk). Here, we do not assume that the database can directly operate on uncompressed data. Recall that $S_{T'}$ is the join-key selectivity of T' (cf. Section 3.4). We use T'' to denote the table further reduced by a Bloom filter $bf_{L'}$ computed from the filtered HDFS table L' with a false-positive rate ϵ. Then T'' can be estimated to contain $|T''| = |T'| \times S_{T'} \times (1+\epsilon)$ records and to be of size $\|T''\| = \|T'\| \times S_{T'} \times (1+\epsilon)$. Similarly, we can estimate $|L'|$, $\|L'\|$, $|L''|$, and $\|L''\|$.

We first provide the formulas for the cost of the DB-side join with Bloom filter (see Figure 1). We use $C_{db}(T, pred_T, proj_T, comp_bf_{T'})$ to represent the cost of applying predicates, projection, and computing the Bloom filter $bf_{T'}$ on T inside the database. The cost of broadcasting the Bloom filter to all the n HDFS nodes (cf. Section 4.3 for the transfer pattern) is captured by $Net_{btwn}(n \times \|bf_{T'}\|)$. Then $C_{hdfs}(L, pred_L, proj_L, appl_bf_{T'})$ represents the cost of applying the predicates, projection, and Bloom filter on the HDFS table L. Subsequently, L'' is transferred to the database side (cf. Figure 6 for the transfer pattern), the cost of which is captured by $Net_{btwn}(\|L''\|)$. Finally, the final join is carried out inside the database, with the cost denoted as $C_{db}(T' \bowtie L'')$. Later in this section, we will describe in more detail how each of these component costs is derived. To summarize, the total cost of the DB-side join can be modeled as

$$\begin{aligned} C(db_side_{bf}) = \; & C_{db}(T, pred_T, proj_T, comp_bf_{T'}) + Net_{btwn}(n \times \|bf_{T'}\|) \\ & + C_{hdfs}(L, pred_L, proj_L, appl_bf_{T'}) + Net_{btwn}(\|L''\|) + C_{db}(T' \bowtie L''). \end{aligned}$$

Similarly, the cost of the DB-side join without Bloom filter can be computed as follows. Here, $C_{db}(T, pred_T, proj_T)$ is the cost of applying predicates and projection on T in the

database, and $C_{hdfs}(L, pred_L, proj_L)$ is the cost of applying predicates and projection on L:

$$C(db_side) = C_{db}(T, pred_T, proj_T) + C_{hdfs}(L, pred_L, proj_L) + Net_{btwn}(\|L'\|) + C_{db}(T' \bowtie L').$$

For the broadcast join (see Figure 2), we first apply predicates and projection on T in the database ($C_{db}(T, pred_T, proj_T)$), and then T' is broadcast to all HDFS nodes, with the cost denoted as $Net_{btwn}(n \times \|T'\|)$. On the HDFS side, predicates and projection are applied on L (the cost is $C_{hdfs}(L, pred_L, proj_L)$). Finally, the join is executed locally on each HDFS node. Note that in the local join denoted as $T' \bowtie_l L'_{/n}$, L' is distributed across all HDFS nodes (hence the notation $L'_{/n}$), whereas T' is replicated on each HDFS node. We use $C_{hdfs}(T' \bowtie_l L'_{/n})$ to represent this cost. Therefore, the total cost of broadcast join is

$$C(broadcast) = C_{db}(T, pred_T, proj_T) + Net_{btwn}(n \times \|T'\|) + C_{hdfs}(L, pred_L, proj_L) + C_{hdfs}(T' \bowtie_l L'_{/n}).$$

In the repartition join with Bloom filter (see Figure 3), again we use $C_{db}(T, pred_T, proj_T, comp_bf_{T'}) + Net_{btwn}(n \times \|bf_{T'}\|)$ to represent the cost of applying predicates and projections on T, and computing and transferring Bloom filter $bf_{T'}$. In addition, T' is also transferred across the network ($Net_{btwn}(\|T'\|)$). Besides applying predicates, projections, and Bloom filter on L, this algorithm also shuffles the filtered table L'' across the HDFS cluster ($Net_{hdfs}(\|L''\|)$). Finally, the local join is performed on each HDFS node ($C_{hdfs}(T'_{/n} \bowtie_l L''_{/n})$). Note that the local join here is different from that in the broadcast join, as T' is also distributed. The total cost of repartition join is as follows:

$$\begin{aligned} C(repart_{bf}) = {} & C_{db}(T, pred_T, proj_T, comp_bf_{T'}) + Net_{btwn}(n \times \|bf_{T'}\|) + Net_{btwn}(\|T'\|) \\ & + C_{hdfs}(L, pred_L, proj_L, appl_bf_{T'}) + Net_{hdfs}(\|L''\|) \\ & + C_{hdfs}(T'_{/n} \bowtie_l L''_{/n}). \end{aligned}$$

The cost of the repartition join without Bloom filter can simply be estimated as

$$C(repart) = C_{db}(T, pred_T, proj_T) + Net_{btwn}(\|T'\|) + C_{hdfs}(L, pred_L, proj_L) + Net_{hdfs}(\|L'\|) + C_{hdfs}(T'_{/n} \bowtie_l L'_{/n}).$$

The first two steps of the zigzag join (see Figure 4) is the same as in the repartition join with Bloom filter. In the third step, zigzag join also computes another Bloom filter $bf_{L'}$ while applying predicates, projection, and $bf_{T'}$ on L. We use $C_{hdfs}(L, pred_L, proj_L, appl_bf_{T'}, comp_bf_{L'})$ to represent this cost. Subsequently, $bf_{L'}$ is broadcast back to each database worker ($Net_{btwn}(m \times \|bf_{L'}\|)$), and L'' is shuffled ($Net_{hdfs}(\|L''\|)$) among the n HDFS nodes. Then the database applies $bf_{L'}$ on the filtered table T', with cost denoted as $C_{db}(T', appl_bf_{L'})$. After that, the further filtered table T'' is transferred to the HDFS side ($Net_{btwn}(\|T''\|)$), before the final join is carried out locally. Note that in the local join, the build side is L'' (cf. Section 4.5); hence, we use $C_{hdfs}(L''_{/n} \bowtie_l T''_{/n})$ to denote this cost. To summarize, the cost of zigzag join is modeled as

$$\begin{aligned} C(zigzag) = {} & C_{db}(T, pred_T, proj_T, comp_bf_{T'}) + Net_{btwn}(n \times \|bf_{T'}\|) \\ & + C_{hdfs}(L, pred_L, proj_L, appl_bf_{T'}, comp_bf_{L'}) \\ & + Net_{btwn}(m \times \|bf_{L'}\|) + Net_{hdfs}(\|L''\|) \\ & + C_{db}(T', appl_bf_{L'}) + Net_{btwn}(\|T''\|) + C_{hdfs}(L''_{/n} \bowtie_l T''_{/n}). \end{aligned}$$

In summary, the costs of all algorithms are (some terms are reorganized)

$$C(db_side_{bf}) = C_{db}(T, pred_T, proj_T, comp_bf_{T'}) + C_{hdfs}(L, pred_L, proj_L, appl_bf_{T'}) + Net_{btwn}(n \times \|bf_{T'}\|) + Net_{btwn}(\|L''\|) + C_{db}(T' \bowtie L'') \quad (1)$$

$$C(db_side) = C_{db}(T, pred_T, proj_T) + C_{hdfs}(L, pred_L, proj_L) + Net_{btwn}(\|L'\|) + C_{db}(T' \bowtie L') \quad (2)$$

$$C(broadcast) = C_{db}(T, pred_T, proj_T) + C_{hdfs}(L, pred_L, proj_L) + Net_{btwn}(n \times \|T'\|) + C_{hdfs}(T' \bowtie_l L'_{/n}) \quad (3)$$

$$C(repart_{bf}) = C_{db}(T, pred_T, proj_T, comp_bf_{T'}) + C_{hdfs}(L, pred_L, proj_L, appl_bf_{T'}) + Net_{btwn}(n \times \|bf_{T'}\|) + Net_{btwn}(\|T'\|) + Net_{hdfs}(\|L''\|) + C_{hdfs}(T'_{/n} \bowtie_l L''_{/n}) \quad (4)$$

$$C(repart) = C_{db}(T, pred_T, proj_T) + C_{hdfs}(L, pred_L, proj_L) + Net_{btwn}(\|T'\|) + Net_{hdfs}(\|L'\|) + C_{hdfs}(T'_{/n} \bowtie_l L'_{/n}) \quad (5)$$

$$C(zigzag) = C_{db}(T, pred_T, proj_T, comp_bf_{T'}) + C_{hdfs}(L, pred_L, proj_L, appl_bf_{T'}, comp_bf_{L'}) + Net_{btwn}(n \times \|bf_{T'}\|) + Net_{btwn}(m \times \|bf_{L'}\|) + Net_{btwn}(\|T''\|) + Net_{hdfs}(\|L''\|) + C_{db}(T', appl_bf_{L'}) + C_{hdfs}(L''_{/n} \bowtie_l T''_{/n}). \quad (6)$$

Computing or applying a Bloom filter on T or L is always coupled with predicate and projection application, and mainly involves very efficient CPU bit-wise operations. This cost is mostly negligible based on our empirical observation. Therefore, we have $C_{db}(T, pred_T, proj_T, comp_bf_{T'}) \approx C_{db}(T, pred_T, proj_T)$ and $C_{hdfs}(L, pred_L, proj_L, appl_bf_{T'}, comp_bf_{L'}) \approx C_{hdfs}(L, pred_L, proj_L, appl_bf_{T'}) \approx C_{hdfs}(L, pred_L, proj_L)$. With this simplification, the first two terms of Equations (1) through (6) become the same. For the purpose of comparing the different algorithms, we can therefore remove the common costs from the cost formulas, resulting in the following equations:

$$\tilde{C}(db_side_{bf}) = Net_{btwn}(n \times \|bf_{T'}\|) + Net_{btwn}(\|L''\|) + C_{db}(T' \bowtie L'') \quad (7)$$

$$\tilde{C}(db_side) = Net_{btwn}(\|L'\|) + C_{db}(T' \bowtie L') \quad (8)$$

$$\tilde{C}(broadcast) = Net_{btwn}(n \times \|T'\|) + C_{hdfs}(T' \bowtie_l L'_{/n}) \quad (9)$$

$$\tilde{C}(repart_{bf}) = Net_{btwn}(n \times \|bf_{T'}\|) + Net_{btwn}(\|T'\|) + Net_{hdfs}(\|L''\|) + C_{hdfs}(T'_{/n} \bowtie_l L''_{/n}) \quad (10)$$

$$\tilde{C}(repart) = Net_{btwn}(\|T'\|) + Net_{hdfs}(\|L'\|) + C_{hdfs}(T'_{/n} \bowtie_l L'_{/n}) \quad (11)$$

$$\tilde{C}(zigzag) = Net_{btwn}(n \times \|bf_{T'}\|) + Net_{btwn}(m \times \|bf_{L'}\|) + Net_{btwn}(\|T''\|) + Net_{hdfs}(\|L''\|) + C_{hdfs}(L''_{/n} \bowtie_l T''_{/n}) + C_{db}(T', appl_bf_{L'}). \quad (12)$$

Next, we provide details of the individual terms in these cost formulas. Let b_{btwn} be the *effective* transfer rate between the database and HDFS (instead of the transmission speed declared on the network switch specification). Then the cost of transferring s bytes between the two is given by $Net_{btwn}(s) = s/b_{btwn}$. With the assumption of a homogeneous HDFS cluster, the cost of shuffling s bytes in the HDFS cluster can be

estimated as $Net_{hdfs}(s) = s/(n \times b_{hdfs})$, where b_{hdfs} is the *effective* node-to-node transfer rate in the HDFS cluster.

For the local joins in the HDFS cluster, the total resource time can be estimated by the average resource time taken by a single HDFS node. As described in Section 4.5, we use a hash-based join algorithm to perform the local joins. Let R denote the *build* table and S denote the *probe* table of the local hash join. We first model the I/O cost. We use M to represent the available memory, and d to represent the sequential I/O speed of the local machine. The hash join algorithm copartitions the two tables, in case the build-side hash tables cannot all fit in memory. If R can fit in memory, that is, $\|R\| < M$, then no I/O cost is incurred. Otherwise, we need to write R partitions that do not fit in memory onto disk and read them back later. In the worst case, all tuples in S can only join with the on-disk portion of the R table; thus, the S table also needs to be written to disk and read back to finish the local join. Therefore, the estimated I/O cost is $2 \times (\|R\| + \|S\| - M)/d$.

In terms of CPU cost, we divide the CPU operations into *build* operations and *probe* operations. We measure the average time taken by a single record to build into the hash tables (we assume that a large number of records are built into the hash tables, and thus the initial overhead of allocating the hash tables can be omitted), denoted as t^b, and the average time by a single record to probe the hash tables, denoted as t^p. Then, the total CPU cost can be estimated as $|R| \times t^b + |S| \times t^p$. In summary, the total cost of the local hash join is modeled as follows:

$$join_h(R, S, M, d, t^b, t^p) = \begin{cases} |R| \times t^b + |S| \times t^p & \text{if} \|R\| < M \\ |R| \times t^b + |S| \times t^p + \frac{2\times(\|R\|+\|S\|-M)}{d} & \text{otherwise.} \end{cases}$$

Therefore, $C_{hdfs}(R \bowtie_l S) = join_h(R, S, M_{hdfs}, d_{hdfs}, t^b_{hdfs}, t^p_{hdfs})$, where M_{hdfs} denotes the size of memory available to the hash join algorithm, d_{hdfs} is the sequential disk bandwidth, t^b_{hdfs} is the per-record time for building hash tables, and t^p_{hdfs} is the per-record time for probing hash tables on each HDFS node.

For $C_{db}(T' \bowtie L'')$, $C_{db}(T' \bowtie L')$, and $C_{db}(T', appl_bf_{L'})$, we need to model the cost of the operations inside a database. They are more complicated, as the costs depend on the execution plans chosen by the database optimizer. We consider most sensible plans that are likely to be chosen by a textbook query optimizer with accurate statistics.

Let's first consider $C_{db}(T', appl_bf_{L'})$, the cost of applying the Bloom filter on T'. T' is an intermediate table after applying predicates and projection on T. We use M_{db} to denote the buffer pool size on each database worker and assume that data is uniformly distributed across all m database workers. If $\|T'\|/m < M_{db}$, T' is likely to be in the buffer pool when we need to apply the Bloom filter, and hence there is no I/O cost. Otherwise, the database will spill T' to disk automatically. So, the I/O cost is $(\|T'\|/m - M_{db})/d_{db}$, where d_{db} is the sequential disk bandwidth for each database worker. Again, the CPU cost of applying the Bloom filter is mostly negligible. Therefore, we can estimate $C_{db}(T', appl_bf_{L'}) = \max\{0, (\|T'\|/m - M_{db})/d_{db}\}$.

For $C_{db}(T' \bowtie L'')$ and $C_{db}(T' \bowtie L')$, the database could choose a broadcast join or a repartition join based on the sizes of the two inputs. Let's use R and S to represent the two tables to be joined. In the case of a broadcast join, either table can be broadcast. When R is broadcast, the network cost is $\|R\|/b_{db}$, where b_{db} is the effective transfer rate between database workers. As for the local joins on each database worker, the query optimizer can choose among different local join algorithms, such as hash join, sort-merge join, and nested loops join. Since both local tables in our case are intermediate results (no indexes on them, and thus nested loops join is unlikely to be used) and are not ordered (thus sort-merge join is unlikely to be chosen either), the most reasonable local join algorithm chosen by the optimizer is a hash join. In fact, hash join

is chosen here by the DB2 optimizer in our prototype implementation. Therefore, using the analysis for the local hash join and assuming the optimizer chooses the smaller table as the build side for the local join, we can estimate the cost of the local join in the database to be $join_h(\min\{R, S_{/m}\}, \max\{R, S_{/m}\}, M_{db}, d_{db}, t^b_{db}, t^p_{db})$, where $\min\{R, S_{/m}\}$ returns the smaller of table R and table $S_{/m}$, which is the local partition of S, and $\max\{R, S_{/m}\}$ returns the larger of the two. On average, the cardinality and size of $S_{/m}$ can be estimated as $|S_{/m}| = |S|/m$ and $\|S_{/m}\| = \|S\|/m$. t^b_{db} and t^p_{db} are the average time to build into and probe the hash tables per record, respectively. In summary, when R is broadcast, the total cost is $\|R\|/b_{db} + join_h(\min\{R, S_{/m}\}, \max\{R, S_{/m}\}, M_{db}, d_{db}, t^b_{db}, t^p_{db})$. Analogously, we can estimate the cost when S is broadcast.

In the case of a repartition join, both tables are shuffled across the database workers, and thus the network cost is $(\|R\| + \|S\|)/(m \times b_{db})$. The cost of the local join is $join_h(\min\{R_{/m}, S_{/m}\}, \max\{R_{/m}, S_{/m}\}, M_{db}, d_{db}, t^b_{db}, t^p_{db})$. Assuming that the optimizer always chooses the best plan, we can estimate the join cost as

$$C_{db}(R \bowtie S) = \min\left\{\frac{\|R\|}{b_{db}} + join_h(\max\{R, S_{/m}\}, \max\{R, S_{/m}\}, M_{db}, d_{db}, t^b_{db}, t^p_{db}),\right.$$
$$\frac{\|S\|}{b_{db}} + join_h(\min\{R_{/m}, S\}, \max\{R_{/m}, S\}, M_{db}, d_{db}, t^b_{db}, t^p_{db}),$$
$$\left.\frac{\|R\| + \|S\|}{m \times b_{db}} + join_h(\min\{R_{/m}, S_{/m}\}, \max\{R_{/m}, S_{/m}\}, M_{db}, d_{db}, t^b_{db}, t^p_{db})\right\}.$$

Now, after expanding all the terms of Equations (7) through (12), we get the final cost formulas. The notations used in the cost formulas are provided in Table II:

$$\tilde{C}(db_side_{bf}) = \frac{n \times \|bf_{T'}\|}{b_{btwn}} + \frac{\|L''\|}{b_{btwn}} + C_{db}(T' \bowtie L'') \tag{13}$$

$$\tilde{C}(db_side) = \frac{\|L'\|}{b_{btwn}} + C_{db}(T' \bowtie L') \tag{14}$$

$$\tilde{C}(broadcast) = \frac{n \times \|T'\|}{b_{btwn}} + join_h\big(T', L'_{/n}, M_{hdfs}, d_{hdfs}, t^b_{hdfs}, t^p_{hdfs}\big) \tag{15}$$

$$\tilde{C}(repart_{bf}) = \frac{n \times \|bf_{T'}\|}{b_{btwn}} + \frac{\|T'\|}{b_{btwn}} + \frac{\|L''\|}{n \times b_{hdfs}} + join_h\big(T'_{/n}, L''_{/n}, M_{hdfs}, d_{hdfs}, t^b_{hdfs}, t^p_{hdfs}\big) \tag{16}$$

$$\tilde{C}(repart) = \frac{\|T'\|}{b_{btwn}} + \frac{\|L'\|}{n \times b_{hdfs}} + join_h\big(T'_{/n}, L'_{/n}, M_{hdfs}, d_{hdfs}, t^b_{hdfs}, t^p_{hdfs}\big) \tag{17}$$

$$\tilde{C}(zigzag) = \frac{n \times \|bf_{T'}\|}{b_{btwn}} + \frac{m \times \|bf_{L'}\|}{b_{btwn}} + \frac{\|T''\|}{b_{btwn}} + \frac{\|L''\|}{n \times b_{hdfs}} + join_h\big(T''_{/n}, L''_{/n}, M_{hdfs}, d_{hdfs}, t^b_{hdfs}, t^p_{hdfs}\big) + \max\left\{0, \frac{\|T'\|/m - M_{db}}{d_{db}}\right\}, \tag{18}$$

Table II. Notations Used in the Cost Formulas

System Parameters

m	# of database workers
n	# of HDFS nodes
M_{db}	buffer pool size of each database worker
M_{hdfs}	memory available for the local hash join on each HDFS node
b_{btwn}	effective network transfer rate between the database and HDFS
b_{db}	effective network transfer rate between database workers
b_{hdfs}	effective network transfer rate between HDFS nodes
d_{db}	sequential I/O speed on a database worker
d_{hdfs}	sequential I/O speed on an HDFS node
t_{db}^{b}	average per-record time for building hash tables for the local hash join in a database worker
t_{db}^{p}	average per-record time for probing hash tables for the local hash join in a database worker
t_{hdfs}^{b}	average per-record time for building hash tables for the local hash join in an HDFS node
t_{hdfs}^{b}	average per-record time for probing hash tables for the local hash join in an HDFS node

Query Parameters

$\|T\|$	cardinality of the database table T
$\|L\|$	cardinality of the HDFS table L
$\\|T\\|$	on-disk size of the database table T
$\\|L\\|$	on-disk size of the HDFS table L
$\\|bf_{T'}\\|$	the size of the database Bloom filter
$\\|bf_{L'}\\|$	the size of the HDFS Bloom filter, $\\|bf_{L'}\\| = \\|bf_{T'}\\|$
λ_T	ration of the uncompressed size of T to the on-disk size of T
λ_L	ration of the uncompressed size of L to the on-disk size of L
σ_T	local predicate selectivity on T
σ_L	local predicate selectivity on L
π_T	reduction factor of the projection on T
π_L	reduction factor of the projection on L
$S_{T'}$	join-key selectivity on T' (T' is T filtered by local predicates and projection)
$S_{L'}$	join-key selectivity on L' (L' is L filtered by local predicates and projection)
ϵ	false-positive rate of a Bloom filter

where

$$C_{db}(R \bowtie S) = \min\left\{ \frac{\|R\|}{b_{db}} + join_h\big(\max\{R, S_{/m}\}, \max\{R, S_{/m}\}, M_{db}, d_{db}, t_{db}^b, t_{db}^p\big), \right.$$
$$\frac{\|S\|}{b_{db}} + join_h\big(\min\{R_{/m}, S\}, \max\{R_{/m}, S\}, M_{db}, d_{db}, t_{db}^b, t_{db}^p\big),$$
$$\left. \frac{\|R\| + \|S\|}{m \times b_{db}} + join_h\big(\min\{R_{/m}, S_{/m}\}, \max\{R_{/m}, S_{/m}\}, M_{db}, d_{db}, t_{db}^b, t_{db}^p\big) \right\}$$

$$|R_{/k}| = \frac{|R|}{k}, \qquad \|R_{/k}\| = \frac{\|R\|}{k}$$

$$join_h(R, S, M, d, t^b, t^p) = \begin{cases} |R| \times t^b + |S| \times t^p & \text{if } \|R\| < M \\ |R| \times t^b + |S| \times t^p + \frac{2\times(\|R\|+\|S\|-M)}{d} & \text{otherwise} \end{cases}$$

$$|T'| = |T| \times \sigma_T, \qquad \|T'\| = \|T\| \times \sigma_T \times \pi_T \times \lambda_T$$
$$|L'| = |L| \times \sigma_L, \qquad \|L'\| = \|L\| \times \sigma_L \times \pi_L \times \lambda_L$$

$$|T''| = |T| \times \sigma_T \times S_{T'} \times (1+\epsilon), \qquad \|T''\| = \|T\| \times \sigma_T \times \pi_T \times \lambda_T \times S_{T'} \times (1+\epsilon)$$
$$|L''| = |L| \times \sigma_L \times S_{L'} \times (1+\epsilon), \qquad \|L''\| = \|L\| \times \sigma_L \times \pi_L \times \lambda_L \times S_{L'} \times (1+\epsilon).$$

5.1. Parameter Estimations

As shown in Table II, the parameters used in the cost formulas can be categorized into *system* parameters and *query* parameters. System parameters are only related to the system environment of the hybrid warehouse and they stay the same for all queries, whereas query parameters change for each query.

Simple system parameters such as m, n, M_{db}, and M_{hdfs} can be simply obtained from the configuration of the database and the execution engine on Hadoop. The other system parameters are related to the hardware and software used. Instead of using the values provided in the hardware specification, we obtain their *effective* values through a learning suite that runs a number of test programs after the system is set up. Inside our own JEN engine, we instrument simple tests to directly measure d_{hdfs}, b_{hdfs}, t^b_{hdfs}, and t^p_{hdfs} on the HDFS side. On the database side, such direct measurements are not applicable, because it is a black box to us. To test the system parameters related to the database, we design a set of queries to infer the values of these parameters.

To learn d_{db} (the sequential I/O speed on a database worker), we measure the query time of reading a table of different sizes with a cold buffer pool. For b_{db} (the effective network transfer rate between database workers), we design two query cases that join two tables. In the first case, we join two tables that are already copartitioned on the join key across the database workers, so the join doesn't incur network I/O at all. In the second case, the first table is partitioned on the join key, whereas the other table is not, and thus it needs to be repartitioned across the network to carry out the join. Based on the size of the second table and the observed performance difference between the two cases, we can estimate b_{db}.

To compute t^b_{db} (the average per-record time for building hash tables), again we use a set of special join queries between two tables. Each join is between a large table and a small table, and is executed using a hash join with the large table as the probe table and the small one as the build table (we verify this by observing the query execution plans produced by the database). The probe table for all the queries are the same, while we create different build tables by maintaining the same joinable records but varying the numbers of nonjoinable records. This way, although each query has a different number of hash-table build operations, they all produce the same join results. By comparing the difference between the build table size and the observed query execution time, we can estimate the per-record build time t^b_{db}. Similarly, by only changing the probe table size but maintaining the same join results, we can estimate the per-record probe time t^p_{db}. Finally, to learn b_{btwn} (the effective network transfer rate between the database and HDFS nodes), we design a database UDF that sends a table through TCP/IP sockets to a test program in JEN.

For query parameters, the Bloom filter size ($\|bf_{T'}\|$ and $\|bf_{L'}\|$) is a fixed value for all the algorithms, and it is very easy to get the values of $\|T\|$ and $\|L\|$ by measuring the file sizes. There is also a good chance that the database has statistics about the cardinality of T, the number of unique values on the predicate columns, and the join column. As a result, it is fairly easy to estimate $|T|$, λ_T, π_T, and σ_T using existing selectivity estimation techniques [Mannino et al. 1988]. Given the chosen size of the Bloom filter and the number of hash functions, we can use the number of unique values in the join column of T to estimate the false-positive rate ϵ of the Bloom filter. For the HDFS table, if there are available statistics (e.g., Hive uses ANALYZE TABLE command to collect table statistics), we can use them to estimate $|L|$, λ_L, σ_L, and π_L. Otherwise, we can resort to a sample from L to estimate them. Finally, the join-key selectivities $S_{T'}$ and $S_{L'}$ can be computed either by using a classical approach of estimating join result sizes [Swami and Schiefer 1994] or through samples from both tables.

6. EXPERIMENTAL EVALUATION

Experimental Setup. For experimental studies, we chose different hardware configurations and cluster setups for DB2 DPF and HDFS, to reflect the fact that an EDW is typically deployed on a small number of high-end servers, whereas an HDFS cluster consists of a larger number of commodity machines. For DB2 DPF, we used five servers. Each had two Intel Xeon CPUs @ 2.20GHz, with six physical cores each (12 physical cores in total), 12 SATA disks, one 10Gbit Ethernet card, and a total of 96GB RAM. Each node runs 64-bit Ubuntu Linux 12.04, with a Linux Kernel version 3.2.0-23. We ran six database workers on each server, resulting in a total of 30 DB2 workers. Eleven out of the 12 disks on each server were used for DB2 data storage. We set the buffer pool size of each DB2 worker to be 10GB. For the HDFS cluster, we used 31 IBM System x iDataPlex dx340 servers. Each consisted of two quad-core Intel Xeon E5540 64-bit 2.8GHz processors (eight cores in total), 32 GB RAM, 5x SATA disks, and interconnected using 1Gbit Ethernet. Each server ran Ubuntu Linux (kernel version 2.6.32-24) and Java 1.6. One server was dedicated as the NameNode, whereas the other 30 were used as DataNodes. We reserved one disk for the OS and the remaining four for HDFS on each DataNode. The HDFS replication factor is set to 2. A JEN worker was run on each DataNode and the JEN coordinator was run on the Namenode. Twenty gigabytes of memory were allocated for each JEN worker for join processing. Finally, the two clusters were connected by a 20Gbit switch. With this experimental setup and measurement through test programs, we obtained the values of the system parameters in the cost model: $m = 30$, $n = 30$, $M_{db} = 10$GB, $M_{hdfs} = 20$GB, $b_{btwn} = 213.5$MB/s, $b_{hdfs} = 67.6$MB/s, $b_{db} = 138.1$MB/s, $d_{hdfs} = 209$MB/s, $d_{db} = 2.1$GB/s, $t^b_{db} = 1.5$ microseconds, $t^p_{db} = 0.5$ microseconds, $t^b_{hdfs} = 3.75$ microseconds, $t^p_{hdfs} = 2.96$ microseconds.

Dataset. We generated synthetic datasets in the context of the example query scenario described in Section 2. In particular, we generated a transaction table T of 97GB with 1.6 billion records stored in DB2 DPF and a log table L on HDFS with about 15 billion records. The log table is about 1TB when stored in text format. We also stored the log table in the Parquet columnar format[7] with Snappy compression[8] to more efficiently ingest data from HDFS. The I/O layer of our JEN workers is able to push down projections when reading from this columnar format. The 1TB text log data is reduced to about 421GB in Parquet format. By default, our experiments were run on the Parquet formatted data, but in Section 6.5, we will compare the Parquet format against text format to study their effect on performance. Note that the 1TB log data in text format is larger than the aggregate memory in the HDFS cluster ($30 \times 32 = 960$GB). In addition, the processing on the Hadoop side is scan based, so the file system caching (with LRU policy) does not eliminate sequential IO for this case.

The schemas of the transaction table and the log table are listed as follows:

```
T(uniqKey BIGINT, joinKey INT, corPred INT, indPred INT,
  predAfterJoin DATE, dummy1 VARCHAR(50), dummy2 INT, dummy3 TIME)

L(joinKey INT, corPred INT, indPred INT, predAfterJoin DATE,
  groupByExtractCol VARCHAR(46), dummy CHAR(8))
```

The transaction table T is distributed on a unique key, called `uniqKey`, across the DB2 workers. The two tables are joined on a 4-byte `INT` field `joinKey`. In both tables, there is one `INT` column correlated with the join key called `corPred`, and another `INT` column

[7]http://parquet.io.
[8]http://code.google.com/p/snappy.

independent of the join key called indPred. They are used for local predicates. The DATE fields, named predAfterJoin, on the two tables are used for the predicate after the join. The VARCHAR column groupByExtractCol in L is used for group-by. The remaining columns in each table are just dummy columns. Values of all fields in the two tables are uniformly distributed. We chose uniform distribution as it is easy to control the predicate and join-key selectivities for experimentation. Our current implementation of the proposed join algorithms does not handle data skews specially. Skew handling methods in parallel joins have been proposed in DeWitt et al. [1992], Poosala and Ioannidis [1996], and Xu et al. [2008]. Both Apache PIG[9] and HIVE[10] systems provide some skew handling mechanisms for performing joins. We can employ some of these techniques to handle skewness in data, but defer this study as a future work.

The query that we ran in our experiments can be expressed in SQL as follows:

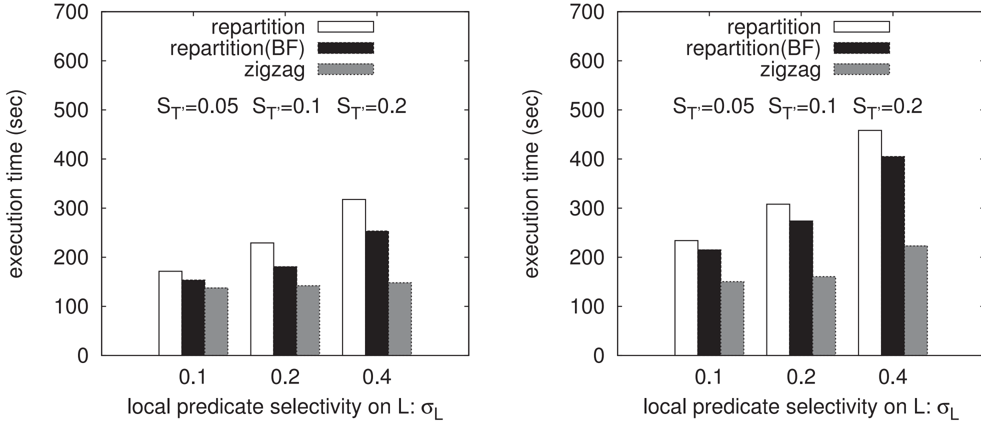
```
SELECT extract_group(L.groupByExtractCol), COUNT(*)
FROM T, L
WHERE T.corPred<=a
  AND T.indPred<=b
  AND L.corPred<=c
  AND L.indPred<=d
  AND T.joinKey=L.joinKey
  AND DAYS(T.predAfterJoin)-DAYS(L.predAfterJoin)>=0
  AND DAYS(T.predAfterJoin)-DAYS(L.predAfterJoin)<=1
GROUP BY extract_group(L.groupByExtractCol)
```

In the previous query, the local predicates on T and L are on the combination of the corPred and the indPred columns, so that we can change the join selectivities given the same selectivities of the combined local predicates. In particular, by modifying constants a and c, we can change the number of join keys participating in the final join from each table, but we can also modify the constants b and d accordingly so that the selectivity of the combined predicates stays intact for each table. We apply a UDF (extract_group) on the VARCHAR column groupByExtractCol to extract an INT column as the group-by column for the final aggregate COUNT(*). To fully exploit the SQL support in DB2, we build one index on (corPred, indPred) and another index on (corPred, indPred, joinKey) of table T. The second index enables calculations of Bloom filters on T using an index-only access plan.

There are 16 million unique join keys in our dataset. We create Bloom filters of 128 million bits (16MB) using two hash functions, which provides a roughly 5% false-positive rate. Note that exploring the different combinations of Bloom filter size and number of hash functions has been well studied before [Bloom 1970] and is beyond the scope of this article. Our particular choice of the parameter values gave us good performance results in our experiments.

Given the aforementioned dataset, the following query parameters used in the cost model were fixed: $|T| = 1646983703$, $\|T\| = 97\text{GB}$, $\lambda_T = 0.75$ (on-disk files contain some extra page layout information), $|L| = 15158236461$, $\|L\| = 421\text{GB}$, $\lambda_L = 2.18$ for Parquet format (on-disk files were columnar and compressed), and $\|L\| = 1\text{TB}$, $\lambda_L = 0.9$ for text format (numerical fields were written as text in the on-disk files). With our example query, $\|bf_{T'}\| = \|bf_{T'}\| = 16\text{MB}$, $\epsilon = 0.05$, $\pi_T = 0.24$, and $\pi_L = 0.87$ were also fixed. In the experiments, we varied the predicate selectivities, σ_T and σ_L, as well as the join-key selectivities, $S_{T'}$ and $S_{L'}$.

[9]https://wiki.apache.org/pig/PigSkewedJoinSpec.
[10]https://cwiki.apache.org/confluence/display/Hive/Skewed+Join+Optimization.



(a) Local predicate selectivity on T: $\sigma_T = 0.1$, join-key selectivity on L': $S_{L'} = 0.1$

(b) Local predicate selectivity on T: $\sigma_T = 0.2$, join-key selectivity on L': $S_{L'} = 0.2$

Fig. 8. Execution time (in seconds) of zigzag join versus repartition joins.

Our experiments were the only workloads that ran on the DPF cluster and the HDFS cluster. But we purposely allocated fewer resources to the DPF cluster to mimic the case that the database is more heavily utilized. For all the experiments, we reported the warm-run performance numbers (we ran each experiment multiple times and excluded the first run when taking the average).

6.1. HDFS-Side Joins

We first study the HDFS-side join algorithms. We start by demonstrating the superiority of our zigzag join to the other repartition-based joins and then investigate when to use the broadcast join versus the repartition-based joins.

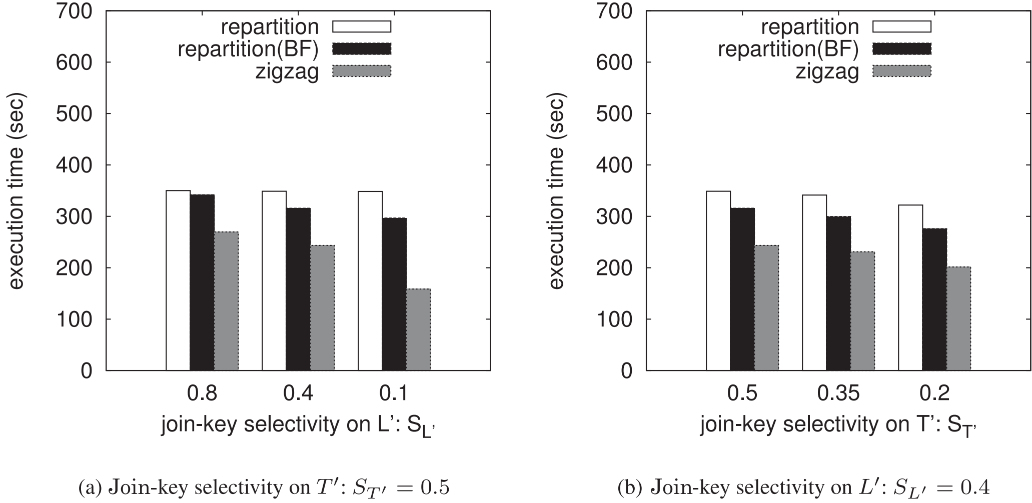
6.1.1. Zigzag Join Versus Repartition Joins. We now compare the zigzag join to the repartition joins both with and without Bloom filter. All three repartition-based join algorithms are best used when local predicates on both the database and HDFS tables are not selective.

Figure 8 compares the execution times of the three algorithms with varying predicate and join-key selectivities on the Parquet formatted log table. It is evident that the zigzag join is the most efficient among all the repartition-based joins. It is up to 2.1× faster than the repartition join without Bloom filter and up to 1.8× faster than the repartition join with Bloom filter. The performance differences among the three algorithms can be explained through their cost formulas in Equations (16), (17), and (18). The size of the Bloom filter (16MB) is so small compared to the filtered L table that the cost of transferring it through the network is almost negligible in this experiment. When ignoring this cost, the repartition join with Bloom filter obviously has a lower cost than its counterpart without Bloom filter. More importantly, zigzag join is the only algorithm that can fully utilize the join-key predicates as well as the local predicates on both sides to reduce the data transferred through the network. For example, when we zoom in on the last column of three bars in Figure 8(a), Table III details the number of HDFS tuples shuffled across the JEN workers as well as the number of database tuples sent to the HDFS side for the three algorithms. The zigzag join is able to cut down the shuffled HDFS data by roughly 10× (corresponding to $S_{L'} = 0.1$) and the transferred database data by around 5× (corresponding to $S_{T'} = 0.2$). Although zigzag

Table III. Data Transferred in Zigzag Join Versus Repartition Joins

	HDFS Tuples Shuffled	DB Tuples Sent
Repartition	5,854 million	165 million
Repartition (BF)	591 million	165 million
Zigzag	591 million	30 million

Local predicate selectivity on T: $\sigma_T = 0.1$, local predicate selectivity on L: $\sigma_L = 0.4$, join-key selectivity on T': $S_{T'} = 0.2$, join-key selectivity on L': $S_{L'} = 0.1$.



(a) Join-key selectivity on T': $S_{T'} = 0.5$

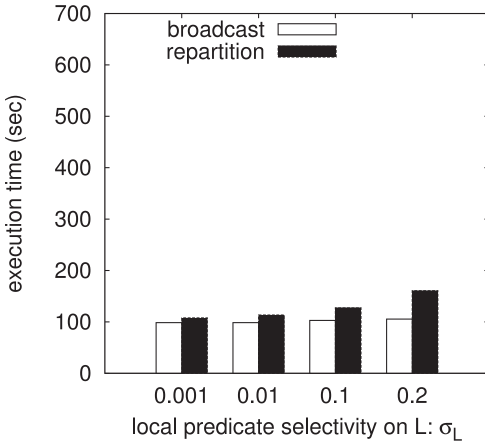
(b) Join-key selectivity on L': $S_{L'} = 0.4$

Fig. 9. Execution time (in seconds) of zigzag join with different join-key selectivities $S_{L'}$ and $S_{T'}$, when local predicate selectivities on T and L are $\sigma_T = 0.1$ and $\sigma_L = 0.4$, respectively.

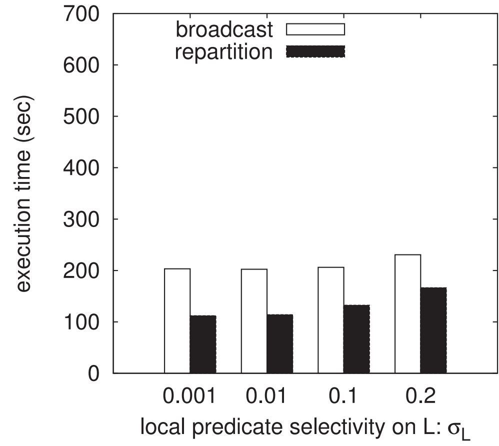
join requires an extra scan of the intermediate database table T' to apply the HDFS Bloom filter, this overhead is well offset by the saved time from the smaller tables joined locally on each HDFS node. In fact, in our experiments, T' is mostly already cached in the database buffer pool, and thus no I/O cost was incurred at all.

In Figure 9, we fix the predicate selectivities $\sigma_T = 0.1$ and $\sigma_L = 0.4$ to explore the effect of different join-key selectivities $S_{L'}$ and $S_{T'}$ on the three algorithms. As expected, with the same size of T' and L', the performance of zigzag join improves when the join-key selectivity $S_{L'}$ or $S_{T'}$ decreases.

6.1.2. Broadcast Join Versus Repartition Join. Besides the three repartition-based joins studied earlier, broadcast join is another HDFS-side join. To find out when this algorithm works best, we compare broadcast join and the repartition join without Bloom filter in Figure 10. We do not include the repartition join with Bloom filter or the zigzag join in this experiment, as even the basic repartition join is already comparable or better than broadcast join in most cases. As shown in the cost formulas in Equations (15) and (17), the tradeoff between the broadcast join and the repartition join is mainly broadcasting T' through the interconnection between the two clusters (the data transferred is $30 \times \|T'\|$ since we have 30 HDFS nodes) versus sending T' once through the interconnection and shuffling L' within the HDFS cluster. Due to the multithreaded implementation described in Section 4.4, the shuffling of L' is interleaved with the reading of L in JEN; thus, this shuffling overhead is somewhat masked by the reading time. As a result, broadcast join performs better only when T' is significantly smaller than L'. In our setting, broadcast join is only preferable when

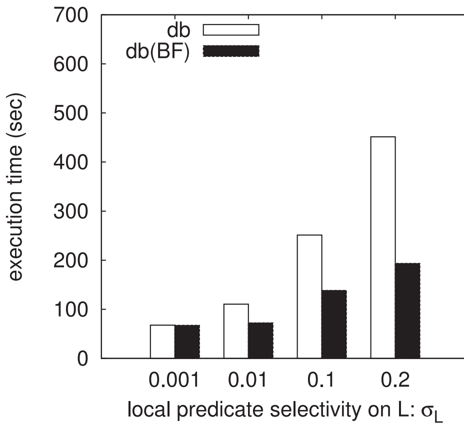


(a) Local predicate selectivity on T: $\sigma_T = 0.001$

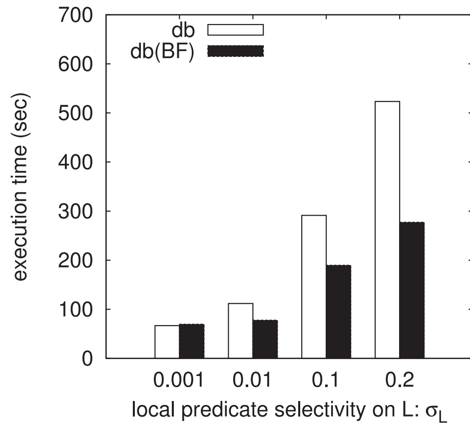


(b) Local predicate selectivity on T: $\sigma_T = 0.01$

Fig. 10. Execution time (in seconds) of broadcast join versus repartition join.



(a) Local predicate selectivity on T: $\sigma_T = 0.05$, join-key selectivity on L': $S_{L'} = 0.05$



(b) Local predicate selectivity on T: $\sigma_T = 0.1$, join-key selectivity on L': $S_{L'} = 0.1$

Fig. 11. Execution time (in seconds) of DB-side joins.

the local predicates on T are highly selective, for example, $\sigma_T \leq 0.001$ ($\|T'\| \leq 25$MB). In comparison, repartition-based joins are more stable, and the zigzag join is the best HDFS-side algorithm in almost all cases.

6.2. DB-Side Joins

We now compare the DB-side joins with and without Bloom filter to study the effect of the Bloom filter. As expected by comparing their costs in Equations (13) and (14), Figure 11 shows that the Bloom filter is very effective in most cases. For fixed local predicates on T (σ_T) and join-key selectivity on L' ($S_{L'}$), the benefit grows significantly as the size of L' increases. Especially for selective local predicates on T, for example, $\sigma_T = 0.05$, the impact of the Bloom filter is more pronounced. When the local predicates on L are very selective (σ_L is very small), for example, $\sigma_L \leq 0.001$, the size of L' is already

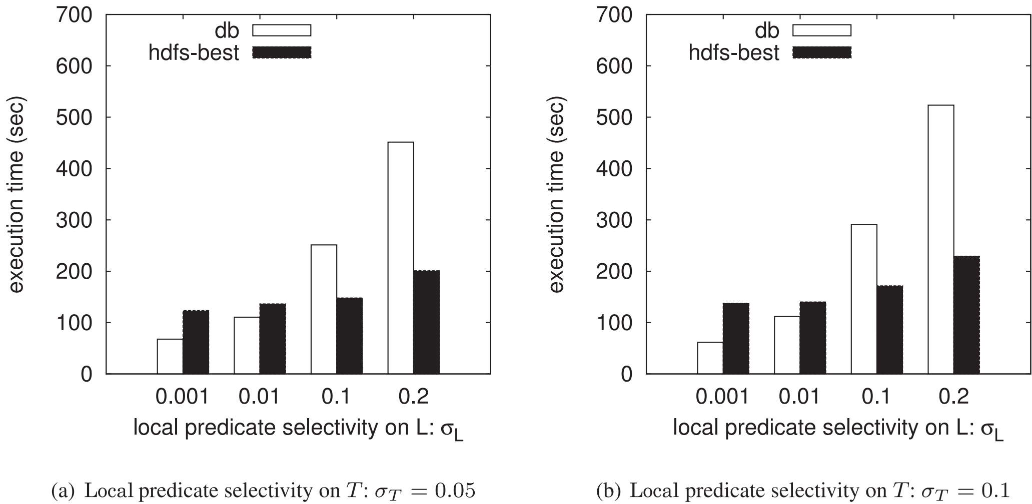


Fig. 12. Execution time (in seconds) of DB-side join versus HDFS-side join without Bloom filter.

very small (e.g., less than 1GB when $\sigma_L = 0.001$), and the overhead of transferring the Bloom filter starts to show up.

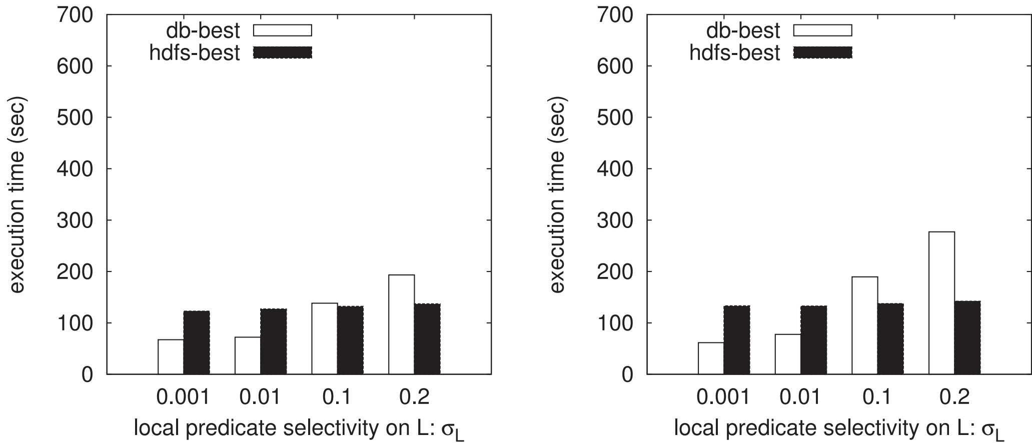
6.3. DB-Side Joins Versus HDFS-Side Joins

Where to perform the final join, on the database side or the HDFS side, is a very important question that we want to address in this article. Most existing solutions [DeWitt et al. 2013; Frazier 2013; McClary 2014] choose to always fetch the HDFS data and execute the join in the database, based on the assumption that SQL-on-Hadoop systems are slower in performing joins. Now that we have better-designed join algorithms and the more sophisticated execution engine in JEN, we want to re-evaluate whether this is the right choice anymore.

We start with the join algorithms without the use of Bloom filters, since the basic DB-side join is used in the existing database/HDFS hybrid systems, and the broadcast join and the basic repartition join are supported in most existing SQL-on-Hadoop systems. Figure 12 compares the DB-side join against the best of the HDFS-side joins (repartition join is the best for all cases in the figure). As shown in this figure, DB-side join performs better only when the predicates on the HDFS table are very selective ($\sigma_L \leq 0.01$). For less selective predicates, probably the common case, the repartition join shows very robust performance, while the DB-side join very quickly deteriorates.

Now, let's also consider all the algorithms with Bloom filters and revisit the comparison in Figure 13. In most of the cases, the DB-side join with Bloom filter is the best DB-side join and the zigzag join is the best HDFS-side join. Comparing this figure to Figure 12, the DB-side join still works better in the same cases as before, although all performance numbers are improved by the use of Bloom filters. The zigzag join shows very steady performance (execution time increases only slightly) with the increase of the L' size, in comparison with the steep deterioration rate of the DB-side join, making this HDFS-side join a more robust choice for joins in the hybrid warehouse.

These experimental results suggest that blindly executing joins in the database is not a good choice anymore. In fact, for common cases when there is no highly selective predicate on the HDFS table, the HDFS-side join is the preferred approach. There are several reasons for this. First of all, the HDFS table is usually much larger than the database table. Even with decent predicate selectivity on the HDFS table, the



(a) Local predicate selectivity on T: $\sigma_T = 0.05$

(b) Local predicate selectivity on T: $\sigma_T = 0.1$

Fig. 13. Execution time (in seconds) of DB-side join versus HDFS-side join with Bloom filter.

Table IV. Validation of the Cost Model

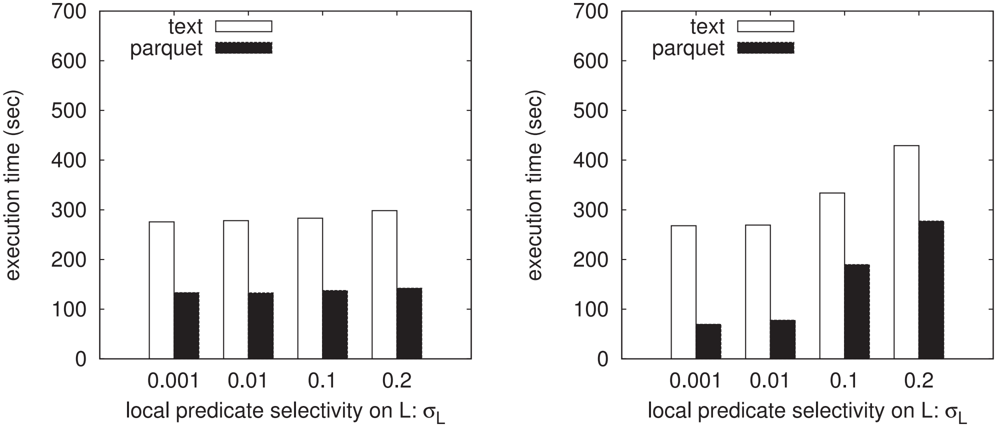
Correspondence to Figures 12 and 13	Selectivity Setting σ_T	σ_L	$S_{T'}$	$S_{L'}$	Best from Cost Model	Best from Experiment	Intersection Metric
1st col in Figures 12&13(a)	0.05	0.001	0.0005	0.05	db(BF)	db(BF)	0
2nd col in Figures 12&13(a)	0.05	0.01	0.005	0.05	db(BF)	db(BF)	0.18
3rd col in Figures 12&13(a)	0.05	0.1	0.05	0.05	zigzag	zigzag	0.08
4th col in Figures 12&13(a)	0.05	0.2	0.1	0.05	zigzag	zigzag	0
1st col in Figures 12&13(b)	0.1	0.001	0.0005	0.1	db(BF)	db(BF)	0
2nd col in Figures 12&13(b)	0.1	0.01	0.005	0.1	db(BF)	db(BF)	0.18
3rd col in Figures 12&13(b)	0.1	0.1	0.05	0.1	zigzag	zigzag	0.14
4th col in Figures 12&13(b)	0.1	0.2	0.1	0.1	zigzag	zigzag	0.06

sheer size after predicates is still big. Second, as our implementation utilizes the DB2 optimizer as is, the HDFS data shipped to the database may need another round of data shuffling among the DB2 workers for the join. Finally, the database side normally has much fewer resources than the HDFS side; thus, when both T' and L' are very large, HDFS-side join should be considered.

6.4. Validation of The Cost Models

In this section, we experimentally validate the cost model of the join algorithms proposed in Section 5. We use the same set of predicate and join-key selectivities as used in Figure 12 and Figure 13. Plugging in these selectivities, we compare the six different join algorithms using the cost formulas in Section 5—db, db(BF), broadcast, repartition, repartition(BF), and zigzag—and check whether the cost model produces the same order of algorithms by their execution times as the “ground truth” from the experiments.

In Table IV, we list the best algorithm found using the cost model (third column) for each selectivity setting, in comparison to the best algorithm found by the experiment (fourth column). In all the cases, the cost model was able to correctly identify the best algorithm! Furthermore, we measure the difference between the ranking of the algorithms based on the cost model and the ranking from the experiments. For this purpose, we employ the distance metric for comparing rankings proposed in Fagin et al. [2003], called the *intersection metric*.



(a) Zigzag join, local predicate selectivity on T: $\sigma_T = 0.1$ (b) db(BF) join, local predicate selectivity on T: $\sigma_T = 0.1$

Fig. 14. Execution time (in seconds) of parquet format versus text format.

The intersection metric between two ranked lists τ_1 and τ_2 of size k is defined as $\delta(\tau_1, \tau_2) = \frac{1}{k}\sum_{i=1}^{k}\delta_i(\tau_1, \tau_2)$, where $\delta_i(\tau_1, \tau_2) = |D_{\tau_1}(i)\Delta D_{\tau_2}(i)|/(2 \times i)$. Here, $D_\tau(i)$ is a set that contains the top i elements from the ranked list τ; $X\Delta Y$ computes the symmetric difference between two sets, that is, $X\Delta Y = (X - Y) \cup (Y - X)$; and $|X|$ denotes the number of elements in the set X. The closer δ is to 0, the more similar two ranked lists are. When $\delta = 0$, the two lists are exactly the same.

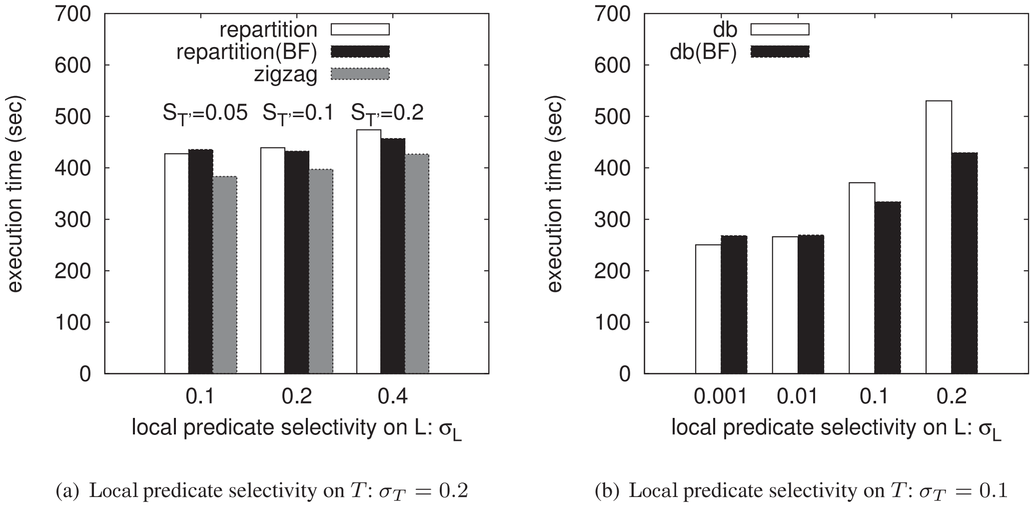
For example, with $\sigma_T = 0.1$, $\sigma_L = 0.2$, $S_{T'} = 0.1$, and $S_{L'} = 0.1$ (last row in Table IV), the cost models rank the algorithms from the best to the worst as follows: *<zigzag, repartition(BF), db(BF), repartition, db, broadcast>*, whereas the empirical experiments produce the ranked list *<zigzag, repartition(BF), repartition, db(BF), db, broadcast>*. To compute the intersection metric, we first look at the top 1 elements of the two lists. They completely agree, and thus $\delta_1 = 0$. Subsequently, we have $\delta_2 = 0$, $\delta_3 = 2/(2 \times 3) = 0.33$, $\delta_4 = 0$, $\delta_5 = 0$, and $\delta_6 = 0$. So, the intersection metric between the two lists is $\delta = (0 + 0 + 0.33 + 0 + 0 + 0)/6 = 0.06$.

As shown in the last column of Table IV, not only does the cost model correctly find the best algorithm in each shown selectivity setting, but also it results in a ranking of algorithms similar to, sometimes even identical to, the real ranking observed from the experiments. The reason the cost model does not always agree with the "ground truth" lies in the fact that the cost formulas simply sum up the different resource times together, whereas in reality multiple resources can be consumed simultaneously, such as the interleaving of disk I/O with network I/O and CPU, due to multithreading (cf. Section 4.4).

6.5. Parquet Format Versus Text Format

We now compare the join performance on the two different HDFS formats. We first pick the zigzag join, which is the best HDFS-side join, and the DB-side join with Bloom filter as the representatives, and show their performance on the Parquet and text formats in Figure 14.

Both algorithms run significantly faster on the Parquet format than on the text format. The 1TB text table on HDFS has already exceeded the aggregated memory size (960GB) of the HDFS cluster; thus, simply scanning the data takes roughly 240 seconds in both cold and warm runs. After columnar organization and compression, the table decreases by about 2.4×, which can now well fit in the local file system cache on each



(a) Local predicate selectivity on T: $\sigma_T = 0.2$ (b) Local predicate selectivity on T: $\sigma_T = 0.1$

Fig. 15. Effect of Bloom filter with text format.

DataNode. In addition, projection pushdown can also be applied when reading from the Parquet format. Therefore, it only takes 38 seconds to read all the required fields from the Parquet data in a warm run. This huge difference in the scanning speed explains the big gap in the performance.

Next, we investigate the effect of using a Bloom filter in joins on the text format. As shown in Figure 15, the improvement due to the Bloom filter is less dramatic on the text format than on the Parquet format. In some cases of the repartition join and the DB-side join, the overhead of computing, transferring, and applying the Bloom filter even outweighs the benefit it brings. Again, the reduced benefit of the Bloom filter is mainly due to the expensive cost of scanning the text format. In addition, there is another reason for the reduced effectiveness of the Bloom filter in the repartition join and the zigzag join. Both algorithms utilize a database Bloom filter to reduce the amount of HDFS data to be shuffled, but with multithreading, the shuffling is interleaved with the scan of the HDFS data (see Section 4.4). For the text format, the reduction of the shuffling cost is largely masked by the expensive scan cost, resulting in the reduced benefit. However, for the zigzag join, with a second Bloom filter to reduce the transferred database data, its performance is always robustly better.

7. DISCUSSION

In this section, we discuss the insights learned from our study.

Among the HDFS-side joins, broadcast join only works for very limited cases, and even when it is better, the advantage is not dramatic. Repartition-based joins are the more robust solutions for HDFS-side joins, and the zigzag join with the two-way Bloom filters always yields the best performance.

A Bloom filter also helps the DB-side join. However, with its steep deterioration rate, the DB-side join works well only when the HDFS table, after applying predicates and projection, is relatively small, and hence its advantages are also confined to limited cases. For a large HDFS table without highly selective predicates, zigzag join is the most reliable join method that works the best most of the time, as it is the only algorithm that fully utilizes the join-key predicates as well as the local predicates on both sides.

Our proposed cost model is able to realistically reflect the relative performance of the join algorithms, correctly finding the best join algorithms under different predicate

and join selectivities. Hence, it can be exploited for query optimization in the context of the hybrid warehouse.

The HDFS data format used significantly affects the performance of a join algorithm. A columnar format with fast compression and decompression techniques provides a dramatic performance boost, compared to the naive text format. So, when data needs to be accessed repeatedly, it is worthwhile to convert the text format into the more advanced format.

Finally, we would like to point out that a major contribution to the good performance of HDFS-side joins is our sophisticated join execution engine on HDFS. It borrows many well-known runtime optimizations from parallel databases, such as pipelining and multithreading. With our careful design in JEN, scanning HDFS data, network communication, and computation are all fully executed in parallel.

8. RELATED WORK

In this article, we study joins in the hybrid warehouse with two fully distributed and independent query execution engines in an EDW and an HDFS cluster, respectively. Although there is a rich literature on distributed join algorithms, most of these existing works study joins in a single distributed system.

In the context of parallel databases, Mackert and Lohman [1986] defined *Bloom join*, which uses Bloom filters to filter out tuples with no matching tuples in a join and achieves better performance than semijoin. Michael et al. [2007] showed how to use a Bloom-filter-based algorithm to optimize distributed joins where the data is stored in different sites. DeWitt and Gerber [1985] studied join algorithms in a multiprocessor architecture and demonstrated that the Bloom filter provided dramatic improvement for various join algorithms. PERF Join [Li and Ross 1995] reduces data transmission of two-way joins based on tuple scan order instead of using Bloom filters. It passes a bitmap of positions, instead of a Bloom filter of values, in the second phase of semijoin. However, unlike Bloom join, it doesn't work well in parallel settings, when there are lots of duplicated values. In Shrinivas et al. [2013], Vertica uses the build-side hash table as a filter to prune out the nonjoinable records while scanning the probe table, when neither table fits in memory. They call this technique sideways information passing. Recently, Polychroniou et al. [2014] proposed track join to minimize network traffic for distributed joins, by scheduling transfers of rows on a per-join-key basis. Determining the desired transfer schedule for each join key, however, requires a full scan of the two tables before the join. Clearly, for systems where scan is a bottleneck, track join would suffer from this overhead.

There has also been some work on join strategies in MapReduce [Blanas et al. 2010; Afrati and Ullman 2011; Lee et al. 2012; Zhang et al. 2012]. Zhang et al. [2012] presented several strategies to build the Bloom filter for large datasets using MapReduce and compared Bloom join algorithms of two-way and multiway joins. In Hive 2.0, a cheap Bloom filter is introduced during the build phase of the Map-side hybrid hash join [Zheng 2015] to help reduce the number of records from the probe table that need to be spilled to disk.

In this article, we also exploit Bloom filters to improve distributed joins, but in a hybrid warehouse setting. Instead of one, our zigzag join algorithm uses two Bloom filters on both sides of the join to reduce the nonjoining tuples. Two-way Bloom filters require scanning one of the tables two times, or materializing the intermediate result after applying local predicates. As a result, two-way Bloom filters are not as beneficial in a single distributed system. But in our case we exploit the asymmetry between HDFS and the database and scan the database table twice. Since the HDFS scan is a dominating cost, scanning the database table twice, especially when we can leverage

indexes, does not introduce significant overhead. As a result, our zigzag join algorithm provides robust performance in many cases.

Because of the need for hybrid warehouses, joins across shared-nothing parallel databases and HDFS have recently received significant attention. Most of the work either simply moves the database data to HDFS, such as Sqoop and the Teradata connector for Hadoop [Teradata 2013], or moves the HDFS data to the database through bulk loading [Teradata 2013; Oracle 2012], external tables [Shrinivas et al. 2013; Oracle 2012], or connectors [Özcan et al. 2011; Teradata 2013]. There are many problems with these approaches. First, HDFS tables are usually pretty big, so it is not always feasible to load them into the database. Second, such bulk reading of HDFS data into the database introduces an unnecessary burden on the carefully managed database resources. Third, database data gets updated frequently, but HDFS still does not support updates properly. Finally, all these approaches assume that the HDFS side does not have proper SQL support that can be leveraged, but this is not true anymore.

Microsoft PolyBase [DeWitt et al. 2013], Pivotal HAWQ [Pivotal 2015], Teradata SQL-H [Frazier 2013], and Oracle Big Data SQL [McClary 2014] all provide online approaches by moving only the HDFS data required for a given query dynamically into the database. They try to leverage both systems for query processing, but only simple predicates and projections are pushed down to the HDFS side. The joins are still evaluated entirely in the database. PolyBase [DeWitt et al. 2013] considers split query processing, but joins are performed on the Hadoop side only when both tables are stored in HDFS.

Hadapt [Bajda-Pawlikowski et al. 2011] also considers splitting query execution between the database and Hadoop, but the setup is very different. As it only uses single-node database servers for query execution, the two tables have to be either prepartitioned or shuffled by Hadoop using the same hash function before the corresponding partitions can be joined locally on each database.

In this article, we show that as the data size grows, it is better to execute the join on the HDFS side, as we end up moving the smaller database table to the HDFS side.

Enabling the cooperation of multiple autonomous databases for processing queries has been studied in the context of federation [Josifovski et al. 2002; Adali et al. 1996; Shan et al. 1995; Tomasic et al. 1998; Papakonstantinou et al. 1995] since the late 1970s. Surveys on federated database systems are provided in Sheth and Larson [1990] and Kossmann [2000]. However, the focus has largely been on schema translation and query optimization to achieve maximum query pushdown into the component databases. Little attention has been paid to the actual data movement between different component databases. In fact, many federated systems still rely on JDBC or ODBC connections to move data through a single data pipe. In the era of big data, even with maximum query push-down, such naive data movement mechanisms result in serious performance issues, especially when the component databases are themselves massive distributed systems. In this article, we provide parallel data movement by fully exploiting the massive parallelism between a parallel database and a join execution engine on HDFS to speed up joins in the hybrid warehouse.

Cost models for distributed query optimization have been well studied in the literature [Mackert and Lohman 1986; Ganguly et al. 1992]. Surveys on existing work are provided in Kossmann [2000] and Chapter 8 of Özsu and Valduriez [2011]. Cost models have also been researched in the context of federation [Roth et al. 1999]. The cost model proposed in this article, in contrast, is for joins in the new environment of hybrid warehouses and captures parallel execution in both databases and Hadoop, as well as the parallel data movement between the two.

Finally, this article is an extension to our prior work in Tian et al. [2015], which proposed the join algorithms for a hybrid warehouse. In particular, we extend the earlier work by introducing a sophisticated cost model for the different join algorithms and empirically validate the cost model through experiments. The cost model constitutes an important building block for a future optimizer, and hence is one step forward toward building a hybrid warehouse architecture.

9. CONCLUSION

In this article, we investigated efficient join algorithms in the context of a hybrid warehouse, which integrates a query engine on HDFS with an EDW. We proposed and implemented various join algorithms and developed their cost models. We showed that it is usually more beneficial to execute the joins on the HDFS side, which is contrary to the prevailing wisdom and commercially available solutions that always execute joins in the EDW. We argue that the best hybrid warehouse architecture should execute joins where the bulk of the data is (after local projection and predicates are applied). In other words, it is better to move the smaller table to the side of the bigger table, whether it is in HDFS or in the database. This hybrid warehouse architecture requires a sophisticated execution engine on the HDFS side and similar SQL capabilities on both sides. Given the recent advancement on SQL-on-Hadoop solutions [Gray et al. 2015; Kornacker et al. 2015; Traverso 2013], we believe this hybrid warehouse solution is now feasible. Our proposed zigzag join algorithm, which performs joins on the HDFS side, utilizing Bloom filters on both sides, is the most robust algorithm that performs well in almost all cases. Finally, our proposed cost model of the join algorithms correctly identifies the best algorithm under different predicate and join selectivities, and thus constitutes an important building block for a future optimizer in the hybrid warehouse environment.

ACKNOWLEDGMENTS

We would like to express our special thanks to Dr. Guy M. Lohman and Dr. Peter J. Haas for their valuable feedback on an early manuscript of this article. We would also like to thank the three anonymous TODS reviewers for their constructive comments that helped us improve this article.

REFERENCES

Sibel Adali, K. Seluk Candan, Yannis Papakonstantinou, and V. S. Subrahmanian. 1996. Query caching and optimization in distributed mediator systems. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD'96)*. 137–146.

Foto Afrati and Jeffrey Ullman. 2011. Optimizing multiway joins in a map-reduce environment. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 23, 9 (2011), 1282–1298.

Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. 2014. The stratosphere platform for big data analytics. *VLDB J.* 23, 6 (2014), 939–964.

Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis Tsotras, Rares Vernica, Jian Wen, Till Westmann, Inci Cetindil, and Madhusudan Cheelangi. 2014. AsterixDB: A scalable, open source BDMS. *PVLDB* 7, 14 (2014), 1905–1916.

Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. 1383–1394.

Kamil Bajda-Pawlikowski, Daniel J. Abadi, Avi Silberschatz, and Erik Paulson. 2011. Efficient processing of data warehousing queries in a split execution environment. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD'11)*. 1165–1176.

Chaitanya Baru, Gilles Fecteau, Ambuj Goyal, Hui-I Hsiao, Anant Jhingran, Sriram Padmanabhan, and Walter Wilson.1995. DB2 parallel edition. *IBM Syst. J.* 34, 2 (April 1995), 292–322.

Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. 2010. A comparison of join algorithms for log processing in MapReduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. 975–986.

Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

Ronnie Chaiken, Bob Jenkins, Paul Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: Easy and efficient parallel processing of massive data sets. *PVLDB* 1, 2 (2008), 1265–1276.

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

David J. DeWitt and Robert H. Gerber. 1985. Multiprocessor hash-based join algorithms. In *Proceedings of the 1985 International Conference on Very Large Data Bases (VLDB'85)*. 151–164.

David J. DeWitt, Alan Halverson, Rimma V. Nehme, Srinath Shankar, Josep Aguilar-Saborit, Artin Avanes, Miro Flasza, and Jim Gramling. 2013. Split query processing in PolyBase. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. 1255–1266.

David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David A. Wood. 1984. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD'84)*. 1–8.

David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. 1992. Practical skew handling in parallel joins. In *Proceedings of the 1992 International Conference on Very Large Data Bases (VLDB'92)*. 27–40.

Ronald Fagin, Ravi Kumar, and D. Sivakumar. 2003. Comparing top K lists. In *Proceedings of the 2003 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'03)*. 28–36.

Doug Frazier. 2013. Dynamic Access: The SQL-H feature for the latest Teradata database leverages data in Hadoop. Retrieved from http://www.teradatamagazine.com/v13n02/Tech2Tech/Dynamic-Access.

Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. 1992. Query optimization for parallel execution. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data (SIGMOD'92)*. 9–18.

Scott C. Gray, Fatma Ozcan, Hebert Pereyra, Bert van der Linden, and Adriana Zubiri. 2015. SQL-on-Hadoop without compromise: How Big SQL 3.0 from IBM represents an important leap forward for speed, portability and robust functionality in SQL-on-Hadoop solutions. Retrieved from http://public.dhe.ibm.com/common/ssi/ecm/sw/en/sww14019usen/SWW14019USEN.PDF.

Vanja Josifovski, Peter Schwarz, Laura Haas, and Eileen Lin. 2002. Garlic: A new flavor of federated query processing for DB2. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*. 524–532.

Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. 2015. Impala: A modern, open-source SQL engine for hadoop. In *Proceedings of the 2015 Conference on Innovative Data Systems Research (CIDR'15)*.

Donald Kossmann. 2000. The state of the art in distributed query processing. *ACM Comput. Surv.* 32, 4 (2000), 422–469.

Taewhi Lee, Kisung Kim, and Hyoung-Joo Kim. 2012. Join processing using bloom filter in MapReduce. In *Proceedings of the 2012 ACM Research in Applied Computation Symposium (RACS'12)*. 100–105.

Zhe Li and Kenneth A. Ross. 1995. PERF join: An alternative to two-way semijoin and bloomjoin. In *Proceedings of the 1995 International Conference on Information and Knowledge Management (CIKM'95)*. 137–144.

Lothar F. Mackert and Guy M. Lohman. 1986. R* optimizer validation and performance evaluation for distributed queries. In *Proceedings of the 1986 International Conference on Very Large Data Bases (VLDB'86)*. 149–159.

Michael V. Mannino, Paicheng Chu, and Thomas Sager. 1988. Statistical profile estimation in database systems. *ACM Comput. Surv.* 20, 3 (1988), 191–221.

Dan McClary. 2014. Oracle Big Data SQL: One Fast Query, All Your Data. Retrieved from https://blogs.oracle.com/datawarehousing/entry/oracle_big_data_sql_one.

Loizos Michael, Wolfgang Nejdl, Odysseas Papapetrou, and Wolf Siberski. 2007. Improving distributed join efficiency with extended bloom filter operations. In *Proceedings of the 2007 International Conference on Advanced Networking and Applications (AINA'07)*. 187–194.

James K. Mullin. 1990. Optimal semijoins for distributed database systems. *TSE* 16, 5 (1990), 558–560.

Oracle. 2012. High Performance Connectors for Load and Access of Data from Hadoop to Oracle Database. Retrieved from http://www.oracle.com/technetwork/bdc/hadoop-loader/connectors-hdfs-wp-1674035.pdf.

Fatma Özcan, David Hoa, Kevin S. Beyer, Andrey Balmin, Chuan Jie Liu, and Yu Li. 2011. Emerging trends in the enterprise data analytics: Connecting Hadoop and DB2 warehouse. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD'11)*. 1161–1164.

M. Tamer Özsu and Patrick Valduriez. 2011. *Principles of Distributed Database Systems* (3rd ed.). Springer.

Yannis Papakonstantinou, Ashish Gupta, Hector Garcia-Molina, and Jeffrey D. Ullman. 1995. A query translation scheme for rapid implementation of wrappers. In *Proceedings of the 1995 International Conference on Deductive and Object-Oriented Databases (DOOD'95)*. 161–186.

Pivotal. 2015. Pivotal HD: HAWQ - A True SQL Engine For Hadoop. Retrieved from http://www.gopivotal.com/sites/default/files/Hawq_WP_042313_FINAL.pdf.

Orestis Polychroniou, Rajkumar Sen, and Kenneth A. Ross. 2014. Track join: Distributed joins with minimal network traffic. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*. 1483–1494.

Viswanath Poosala and Yannis E. Ioannidis. 1996. Estimation of query-result distribution and its application in parallel-join load balancing. In *Proceedings of the 1996 International Conference on Very Large Data Bases (VLDB'96)*. 448–459.

Mary Tork Roth, Fatma Özcan, and Laura M. Haas. 1999. Cost models DO matter: Providing cost information for diverse data sources in a federated system. In *Proceedings of the 1999 International Conference on Very Large Data Bases (VLDB'99)*. 599–610.

Ming-Chien Shan, Rafi Ahmed, Jim Davis, Weimin Du, and William Kent. 1995. Pegasus: A heterogeneous information management system. In *Modern Database Systems*, Won Kim (Ed.). ACM Press/Addison-Wesley Publishing, 664–682.

Amit P. Sheth and James A. Larson. 1990. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.* 22, 3 (1990), 183–236.

Lakshmikant Shrinivas, Sreenath Bodagala, Ramakrishna Varadarajan, Ariel Cary, Vivek Bharathan, and Chuck Bear. 2013. Materialization strategies in the vertica analytic database: Lessons learned. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE'13)*. 1196–1207.

Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. 2014. Orca: A modular query optimizer architecture for big data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*. 337–348.

Arun Swami and K. Bernhard Schiefer. 1994. On the estimation of join result sizes. In *Proceedings of the 1994 International Conference on Extending Database Technology (EDBT'94)*. 287–300.

Teradata. 2013. Teradata Connector for Hadoop. Retrieved from http://developer.teradata.com/connectivity/articles/teradata-connector-for-hadoop-now-available.

Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: A warehousing solution over a map-reduce framework. *PVLDB* 2, 2 (2009), 1626–1629.

Yuanyuan Tian, Tao Zou, Fatma Ozcan, Romulo Goncalves, and Hamid Pirahesh. 2015. Joins for hybrid warehouses: Exploiting massive parallelism in hadoop and enterprise data warehouses. In *Proceedings of the 2015 International Conference on Extending Database Technology (EDBT'15)*. 373–384.

Anthony Tomasic, Louiqa Raschid, and Patrick Valduriez. 1998. Scaling access to heterogeneous data sources with DISCO. *TKDE* 10, 5 (1998), 808–823.

Martin Traverso. 2013. Presto: Interacting with petabytes of data at Facebook. Retrieved from https://www.facebook.com/notes/facebook-engineering/presto-interacting-with-petabytes-of-data-at-facebook/10151786197628920.

Yu Xu, Pekka Kostamaa, Xin Zhou, and Liang Chen. 2008. Handling data skew in parallel joins in shared-nothing systems. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*. 1043–1052.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 2012 USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. 15–28.

Changchun Zhang, Lei Wu, and Jing Li. 2012. Optimizing distributed joins with bloom filters using MapReduce. In *Proceedings of the 2012 International Conference on Computer Applications for Graphics, Grid Computing, and Industrial Environment*. 88–95.

Wei Zheng. 2015. Hybrid Hybrid Grace Hash Join, v1.0. Retrieved from https://cwiki.apache.org/confluence/display/Hive/Hybrid+Hybrid+Grace+Hash+Join,+v1.0#HybridHybridGraceHashJoin,v1.0-BloomFilter.

Received August 2015; revised March 2016; accepted July 2016