Advanced Join Strategies for Large-Scale Distributed Computation

Nicolas Bruno
Microsoft Corp.
nicolasb@microsoft.com

YongChul Kwon
Microsoft Corp.
yongchul.kwon@microsoft.com

Ming-Chuan Wu
Microsoft Corp.
ming-chuan.wu@microsoft.com

ABSTRACT

Companies providing cloud-scale data services have increasing needs to store and analyze massive data sets (e.g., search logs, click streams, and web graph data). For cost and performance reasons, processing is typically done on large clusters of thousands of commodity machines by using high level scripting languages. In the recent past, there has been significant progress in adapting well-known techniques from traditional relational DBMSs to this new scenario. However, important challenges remain open. In this paper we study the very common join operation, discuss some unique challenges in the large-scale distributed scenario, and explain how to efficiently and robustly process joins in a distributed way. Specifically, we introduce novel execution strategies that leverage opportunities not available in centralized scenarios, and others that robustly handle data skew. We report experimental validations of our approaches on Scope production clusters, which power the Applications and Services Group at Microsoft.

1. INTRODUCTION

An increasing number of companies rely on the results of massive data analytics for critical business decisions. Such analysis is crucial to improve service quality, support novel features, and detect changes in patterns over time. Usually the scale of the data volume to be stored and processed is so large that traditional, centralized DBMS solutions are no longer practical. Several companies have thus developed distributed data storage and processing systems on large clusters of thousands of shared-nothing commodity servers [2, 4, 11, 24].

In the MapReduce model, developers provide *map* and *reduce* functions in procedural languages like `Java` or `C#`, which perform data transformation and aggregation. The underlying runtime system achieves parallelism by partitioning the data and processing each partition concurrently, handling load-balancing, outlier detection, and failure recovery. Newer scripting languages [15, 19, 24] raise the level of abstraction of pure MapReduce jobs. These languages offer a single machine programming abstraction and allow developers to focus on application logic, while providing systematic optimizations for the underlying distributed computation.

Over the years, the underlying distributed engines benefited from retrofitting well-known techniques from centralized relational DBMSs. For instance, query optimization and several execution alternatives in distributed scenarios are adapted and generalized from those in traditional database systems. One of the most common operations over data sources is that of joining on a set of attributes. Fortunately, many of the lessons learned over the years for optimizing and executing join queries can be directly translated to the distributed world. At the same time, some unique characteristics of distributed scenarios result in new challenges and opportunities for efficient and reliable join processing. Although the problems and solutions described in this work are generally applicable, we focus on the Scope system, the *cloud-scale computation environment* in Microsoft Applications and Services Group, to showcase our techniques.

Existing taxonomies for join alternatives pivot around join algorithms (e.g., hash-based join), join types (e.g., left-outer join), and in case of n-ary joins, tree shapes (e.g., left-deep joins). In the context of distributed systems, a new dimension to characterize join operations emerges from considering the execution graph topology, which results in new processing alternatives. In this work we study this topic in detail.

In many distributed applications, some values are often extremely more frequent than others[1] and a naïve join strategy over those values typically explodes the amount of intermediate results produced by a single compute node. The cost of processing these highly skewed intermediate results, in turn, either eventually fail the entire query or dominate its latency. For that reason, handling data skew in a robust way is an important problem due to not only the sheer volume of data but also extreme heavy-tail distributions. In this work we introduce different alternatives to deal with such large data skew in a robust way.

The rest of the paper is structured as follows. In Section 2 we review necessary background on the Scope system. In Section 3 we study the solution space for join processing and identify new alternatives in the distributed scenario. In Section 4, we present a class of logical transformations

[1]For instance, in the context of Microsoft Applications and Services, consider the destination IP addresses under denial of service attack, daily queries to the Bing search engine, or a tweet from a celebrity retweeted by millions of followers.

that mitigate the impacts of data skew in distributed join operations. In Section 5 we report experimental evaluations of the techniques introduced in this paper on real workloads in production clusters. Finally, in Section 6 we review related work.

2. THE SCOPE SYSTEM

In this section we describe the main architecture of the SCOPE computation system at Microsoft [11, 24].

2.1 The Language and Data Model

The SCOPE language is declarative and intentionally reminiscing SQL. The select statement is retained along with joins variants, aggregation, and set operators. Like SQL, data is modeled as sets of rows composed of typed columns, and every rowset has a well-defined schema. At the same time, the language is highly extensible and is seamlessly integrated with the .NET framework. Users can easily define their own functions and implement their own versions of relational operators: *extractors* (parsing and constructing rows from a raw file), *processors* (row-wise processing), *reducers* (group-wise processing), *combiners* (combining rows from two inputs), and *outputters* (formatting and outputting final results). These user-defined operators are well-integrated into the system and allow algebraic transformation just like other relational operators. This extensibility allows users to solve problems that cannot be easily expressed in SQL, while at the same time enables sophisticated optimization of scripts.

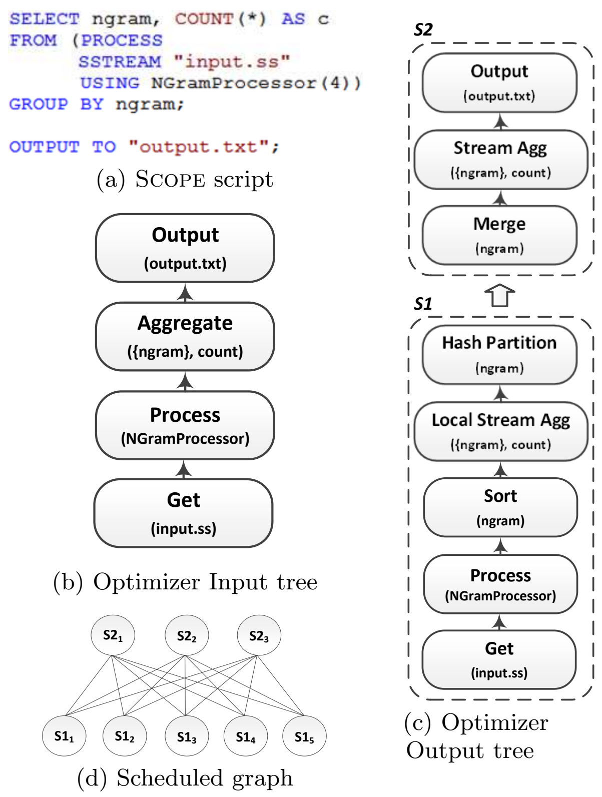
In addition to unstructured data, SCOPE supports *structured streams*. Like tables in a database, a structured stream has a well-defined schema that every record follows. A structured stream is self-contained and includes, in addition to the data itself, rich metadata information such as schema, structural properties (i.e., partitioning and sorting information), statistical information on data distributions, and data constraints [24].

Figure 1(a) shows a simple SCOPE script that counts the different 4-grams of a given single-column structured stream. In the figure, `NGramProcessor` is a C# user-defined operator that outputs, for each input row, all its n-grams (4 in the example). Conceptually, the intermediate output of the processor is a regular rowset that is processed by the main outer query (note that intermediate results are not necessarily materialized between operators at runtime).

2.2 Query Compilation and Optimization

A SCOPE script goes through a series of transformations before it is executed in the cluster. Initially, the SCOPE compiler parses the input script, unfolds views and macro directives, performs syntax and type checking, and resolves names. The result of this step is an annotated abstract syntax tree, which is passed to the query optimizer. Figure 1(b) shows an input tree for the sample script.

The SCOPE optimizer is a cost-based transformation engine that generates efficient execution plans for input trees. Since the language is heavily influenced by SQL, SCOPE is able to leverage existing work on relational query optimization and perform rich and non-trivial query rewritings that consider the input script in a holistic manner. The optimizer returns an execution plan that specifies the steps that are required to efficiently execute the script. Figure 1(c) shows the output



```
SELECT ngram, COUNT(*) AS c
FROM (PROCESS
      SSTREAM "input.ss"
      USING NGramProcessor(4))
GROUP BY ngram;

OUTPUT TO "output.txt";
```

(a) SCOPE script

(b) Optimizer Input tree

(c) Optimizer Output tree

(d) Scheduled graph

Figure 1: Compiling and executing in SCOPE.

from the optimizer, which defines specific implementations for each operation (e.g., stream-based aggregation), data partitioning operations (e.g., the partition and merge operators), and additional implementation details (e.g., the initial sort after the processor, and the unfolding of the aggregate into a local/global pair).

The backend compiler then generates code for each operator and combines a series of operators into an execution unit (or *stage*) obtained by splitting the output tree into components that would be processed by a single compute node. The output of the compilation of a script thus consists of (i) a graph definition file that enumerates all stages and the data flow relationships among them, and (ii) the assembly itself, which contains the generated code. This package is sent to the cluster for execution. The dotted lines in Figure 1(c) depict the two stages corresponding to the input script.

2.3 Job Scheduling and Runtime

The execution of a SCOPE script (or job) is coordinated by a Job Manager (JM). The JM is responsible for constructing the job graph and scheduling work across available resources in the cluster. As described above, a SCOPE execution plan consists of a directed acyclic graph of stages that can be scheduled and executed on different machines independently. A stage represents multiple instances, or *vertices*, which operate over different partitions of the data (see Figure 1(d)). The JM maintains the job graph and keeps track of the state and history of each vertex in the graph. When all inputs of a vertex become ready, the JM considers the vertex *runnable* and places it in a scheduling queue. The actual vertex scheduling order is based on vertex priority and resource availability. One scheduling principle is based on *data locality*. That is, the JM tries to schedule the vertex on a machine that stores or is close to its input whenever possible. If the selected machine is temporarily overloaded, the JM may schedule the vertex to another machine that is close in network topology so reading the input can be done efficiently with minimum network traffic. Additionally, in order to cope with potential hardware failures and variations in machine loads in a large cluster of commodity hardware, JM judiciously launches duplicate vertices, in what is known as speculative *duplicate execution*. Duplicate execution can help improve job latency and job runtime predictability in face of unexpected environmental events in the cluster.

During execution, a vertex reads inputs either locally or remotely. Operators within a vertex are processed in a pipelined fashion, similar to a single-node database engine. Every vertex is given enough memory to satisfy its requirements (e.g., hash tables or external sorts), up to a fraction of total available memory, and a fraction of the available processors. This procedure sometimes prevents a new vertex from running immediately on a busy machine. Similar to traditional database systems, each machine uses admission control techniques and queues outstanding vertices until the required resources are available. The final results of a vertex are written to local disks (non-replicated for performance reasons), waiting for the next vertex to *pull* them.

The *pull* execution model and materialization of intermediate results have many advantages in the context of highly distributed computation. First, it does not require both producer and consumer vertices to run concurrently, which greatly simplifies job scheduling. Second, in case of failures, which are inevitable in a large cluster, all that is required is to rerun the failed vertex from the cached inputs. Only a small portion of a query plan may need to be re-executed. Finally, writing intermediate results frees system memory to execute other vertices and simplifies computation resource management.

3. JOIN PROCESSING

In this section we first briefly review the taxonomy of join processing strategies, and then describe additional join alternatives in a distributed environment.

Traditionally, join strategies are classified in the following dimensions:

- **Join types**, which characterize the semantics of the join, and include cross join (cartesian product), inner join (equi-join, natural join), outer join (left, right, and full outer join) and semi-join (left and right semi-join).

- **Join algorithms**, which are different ways to implement a logical join operator, and include nested-loop joins, sort-based joins, and hash-based joins. In addition, auxiliary data structures, such as *secondary indexes*, *join indexes*, *bitmap indexes*, *bloom filters*, are introduced to further improve the basic join algorithms (*e.g.*, indexed loop join, indexed sort-merge join, and distributed semi-join).

- **Join tree shapes**, which are relevant when performing multiple joins, and characterize the shape of the join tree, such as left-deep, right-deep, and bushy trees.

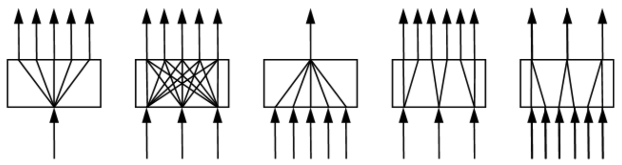


Figure 2: Different types of data exchange topologies (from left to right: initial partitioning, repartitioning, full merge, partial repartitioning, and partial merge).

In a distributed environment, an additional dimension is introduced into the join taxonomy: the *join graph topology*. Graph topologies specify how different partitions of data are processed in a distributed way, and is affected by the following factors [25]:

- **Partitioning schemes**, which characterize how data is partitioned in the system. Partitioning schemes consist of a partition function (e.g., *hash partitioning*, *range partitioning*, *random partitioning* [2], and custom partitioning), a partition key, and a partition count.

- **Data exchange operators**, which modify the partitioning scheme of a data set, and include *initial partitioning*, *repartitioning*, *full merge*, *partial repartitioning* and *partial merge* (see Figure 2).

- **Merging schemes**, which modify data exchange operators by ensuring certain additional intra-partition properties (e.g., order) and include *random merge*, *sort merge*, *concat-merge* and *sort concat-merge* [25].

- **Distribution policies**, which dictate whether partitions can be duplicated to multiple execution nodes, and include *distribution with duplication* and *distribution without duplication*.

Figure 3 illustrates the taxonomy of join processing. Given any combinations along those dimensions, it projects onto the hyper-plane of join implementation strategies. In the rest of this section we will discuss various join implementation strategies for distributed environments in detail. Specifically, we focus on the interactions between join graph topologies and join types in Section 3.1, and how interesting properties affect choices of different join graph topologies and join algorithms in Section 3.2. The join-order dimension is left out for simplicity without loss of generality.

3.1 Join Graph Topologies

Join graph topologies are classified into *symmetric* and *asymmetric* join graphs. In symmetric join graphs, both inputs are horizontally partitioned by the very same partitioning scheme, and the joins are carried out by n compute nodes, where n is the number of partitions. Each compute node reads the pair of the i-th partitions from both inputs and performs the join, thus also called *pair-wise* join graph. The serial join graph is a special case of the symmetric join graph, with $n = 1$.

Asymmetric join graphs are joins on inputs with different partitioning schemes. Figure 4 illustrates a sample asymmetric join graph, where table T is partitioned into five partitions,

[2]Random partitioning can be viewed as a partition function over the key column.

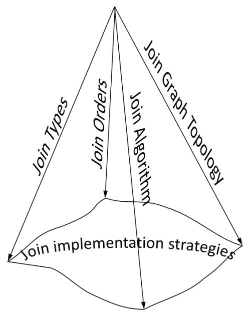


Figure 3: Taxonomy of join alternatives.

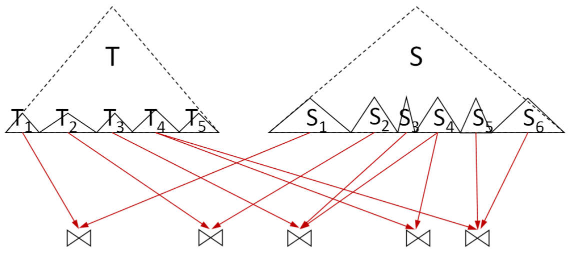


Figure 4: An example of an asymmetric join graph.

while table S is partitioned into six partitions. The join operation is carried out with degree of parallelism (DOP) equal to five. It is asymmetric because (*i*) some partitions participate in more than one join node (those partitions, such as T_4 and S_4, are distributed with replication), and (*ii*) some joins take asymmetric numbers of inputs (*e.g.*, both the third and the fifth join nodes from the left take one input partition from T and two input partitions from S). In addition, not all partitions from T participate in the join, *e.g.*, T_5[3].

It is important to note that join graph topologies describe how partitions from both inputs are connected to the join operator, but are orthogonal to specific join algorithms.

A commonly adopted asymmetric join strategy in a distributed environment is the *broadcast* join graph. In a broadcast join graph, one input is fully-merged to a single partition, while the other input is arbitrarily partitioned. The join is then implemented by broadcasting (with replication by definition) the serial input to all the partitions of the other join operand.

Another common asymmetric join graph is the *full-cross* join graph, where both join operands are arbitrarily partitioned. The join is implemented by broadcasting every left partitions (of the left input) to all the right partitions (of the right input), resulting in a *complete bipartite graph*.

Figure 5 illustrates another generic join graph for dealing with data skew. Both tables T and S are custom partitioned into 4 groups of partitions. The first group of partitions (shown in blue) are connected as pair join graph. The second group of partitions (shown in brown) are connected as right-broadcast graph, and the third group (shown in red) are

[3]Careful readers may question how Figure 4 can be a valid join implementation strategy that produces correct results. Such a join graph is only viable if the system has rich knowledge about distributions and semantics/constraints of the underlying data.

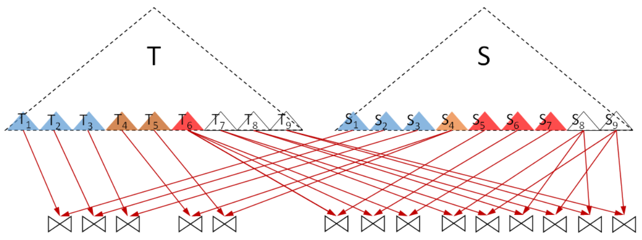


Figure 5: An example of a skew join graph.

connected as left-broadcast graph[4]. Finally, the fourth group (shown in white) are connected as full-cross graph. More details on such strategies are discussed in Section 4.

Not all alternatives in the generalized join taxonomy make intuitive sense. For example, it is commonly less efficient to implement an inner join using random partitioning with a nested loop algorithm (resulting in a full-cross join graph) than using hash partitioning with sort-merge algorithm (resulting in a symmetric join graph) due to the large amount of data transfer in the first alternative. However, in certain cases those intuitively uninteresting approaches become favorable solutions. For example, if extreme data skew is present (either in the inputs or in the join result), the traditional symmetric parallel join graph will suffer on those *outlier* compute nodes (more details in Section 4). Also, in the presence of *interesting properties* [5], full-cross join graphs become interesting parallel join alternatives that have not been considered in traditional DBMSs (see Section 3.2).

If we take the join type dimension into consideration, not all join types can be implemented naïvely with arbitrary join graph topologies, especially those strategies that imply distribution with replication. For instance, a left-outer join cannot be implemented by a left-broadcast join graph without taking additional actions to eliminate duplicates. In Section 4, we will discuss the duplicate elimination techniques used in SCOPE and existing techniques in the literature in Section 6.

Figure 6 depicts the solution space by enumerating the dimensions of the partition functions and the partition counts. The x-axis denotes the partition numbers of the left join operand, the y-axis denotes the partition numbers of the right join operand and the z-axis denotes the partition functions. The z-axis, *i.e.*, $x = y = 1$, represents serial join graphs. The plane where $x = y$ represents symmetric join graphs. All other space represents asymmetric join graphs which can be further classified as follows. The $\{x, z\}$ plane represents the right-broadcast join graphs, and the $\{y, z\}$ plane represents the left-broadcast join graphs. The remaining space represents the full-cross or partial-cross join graphs.

The solution space of inner joins covers the entire space depicted in Figure 6. However, without advanced duplicate elimination techniques, left outer/semi/anti-semi joins can only be covered by the planes where $x = y$ or $y = 1$, and right outer/semi/anti-semi joins can only be covered by the plans where $x = y$ or $x = 1$. Full outer joins can only be covered by the plane where $x = y$, and cartesian products can be covered by the entire space except the plane $x = y$.

[4]A right-broadcast join graph is the one that broadcasts the right input to the left input. Similarly, a left-broadcast join graph is the one that broadcasts the left input to the right input.

[5]Interesting properties include sorting, grouping, partitioning properties and additional access paths.

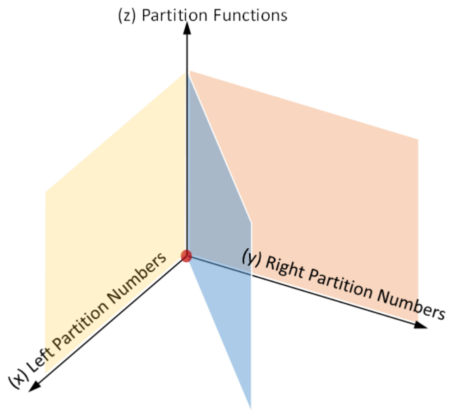


Figure 6: Parallel join graph topologies.

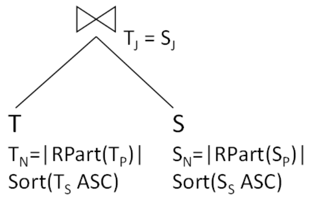


Figure 7: Join example.

Notation	Definition
$\bowtie$	inner (equi-) join operator
$\mathsf{RPart}(C)$	range partition on column sequence C
$\mathsf{Sort}(C)$	sorting on column sequence C
$\mathsf{T_P}$, $\mathsf{S_P}$	partition keys of table T and S, respectively
$\mathsf{T_S}$, $\mathsf{S_S}$	sort keys of table T and S, respectively
$\mathsf{T_J}$, $\mathsf{S_J}$	join keys from table T and S, respectively
$\mathsf{T_N}$, $\mathsf{S_N}$	partition counts of table T and S, respectively
$\lvert\chi\rvert$	degree of parallelism (DOP) on χ, *e.g.*, $\lvert \bowtie \rvert$ denotes the DOP of the join, and $\lvert\mathsf{T}\rvert$ denotes the partition count of table T
$A \sqsubseteq B$	column sequence A is a prefix subset of column sequence B
$A \subseteq B$	column set A is a subset of column set B

Table 1: Notations used in the running example.

3.2 Interesting Properties

In this section, we discuss how interesting properties, such as partitioning and sorting properties, further widen the viable solution space in practice, and how we can exploit such information to improve parallel joins.

Consider the following running example in Figure 7 with notations defined in Table 1. For simplicity, we do not enumerate the entire solution space but instead focus on a typical example. Figure 7 shows a query joining table T with S on an equality predicate $\mathsf{T_J} = \mathsf{S_J}$, where the join is implemented as indexed loop join[6]. Table T is range partitioned on columns $\mathsf{T_P}$ and sorted on $\mathsf{T_S}$. Table S is range partitioned on columns $\mathsf{S_P}$ and sorted on $\mathsf{S_S}$. Since the join algorithm is indexed loop join, we assume that the sorting key $\mathsf{S_S}$ can provide seeks over a subset of the join columns $\mathsf{S_J}$. That is, $\exists\, \mathsf{S'} \sqsubseteq \mathsf{S_S}$, such that $\mathsf{S'} \subseteq \mathsf{S_J}$. Table 2 summarizes various join graph topologies in the distributed environment, given conditions on the interesting properties.

[6]Similar analysis can be applied to other join algorithms with slight modifications.

DOP of the join	Join Graph Topology	Conditions
$\lvert\bowtie\rvert = 1$	serial join graph	$\lvert\mathsf{T}\rvert = \lvert\mathsf{S}\rvert = 1$
$\lvert\bowtie\rvert = \lvert\mathsf{S}\rvert$	left-broadcast join graph	$\lvert\mathsf{T}\rvert = 1$
$\lvert\bowtie\rvert = \lvert\mathsf{S}\rvert \times k$	• for $k = 1$, symmetric join graph (pair-wise join) • for $k < 1$, symmetric join graph with partial merge • for $k > 1$, symmetric join graph with partial repartition	$\lvert\mathsf{T}\rvert > 1$, $\lvert\mathsf{S}\rvert > 1$ and $\mathsf{S_P} \sqsubseteq \mathsf{S_J}$
$\lvert\bowtie\rvert = \lvert\mathsf{T}\rvert \times k$	• for $k = \frac{1}{\lvert\mathsf{T}\rvert}$, serial join • for $k = 1$ and $\lvert\mathsf{S}\rvert = 1$, right-broadcast join graph • for $k \geq 1$ and $\lvert\mathsf{S}\rvert > 1$, full-cross join graph with partial merge on S	$\lvert\mathsf{T}\rvert > 1$ and $\mathsf{S_P} \sqsubseteq \mathsf{S_J} \sqsubseteq \mathsf{S_S}$
$\lvert\bowtie\rvert = \lvert\mathsf{T}\rvert \times \lvert\mathsf{S}\rvert$	full-cross join graph	none

Table 2: Common parallel join graph topologies in distributed environments.

The first three cases are common parallel join strategies in the distributed environment. Two variants of the symmetric join graphs with partial repartitioning and/or partial merge (the last two variations of the third case) are considered by the SCOPE system but have not been described in the literature to the best of our knowledge. For example, if $\mathsf{T_P} \sqsubseteq \mathsf{T_J}$ and $\mathsf{S_P} \sqsubseteq \mathsf{S_J}$, but $\mathsf{T_P} \neq \mathsf{S_P}$ or $\mathsf{T_N} \neq \mathsf{S_N}$, SCOPE will try to refine $\mathsf{RPart}(\mathsf{T_P})$ to $\mathsf{T_{N'}} = \lvert\mathsf{RPart}(\mathsf{T_{P'}})\rvert$ such that $\mathsf{T_{P'}} = \mathsf{S_P}$ and $\mathsf{T_{N'}} = \mathsf{S_N}$, or vice versa. The refinement of the partitioning may include splitting some partitions, or merging some other partitions.

For the fourth case, *i.e.*, $\lvert\bowtie\rvert = \lvert\mathsf{T}\rvert \times k$, the first two variants are also commonly adopted strategies described before. The third variant, where $k \geq 1$ and $\lvert\mathsf{S}\rvert \geq 1$, is a special case of the fifth case (the full-cross join graph) which will be discussed now. The full-cross join graphs for equi-joins are the ones that are less explored in traditional database systems. It is usually less efficient due to the large data volume of the intermediate results, and as such it is pruned heuristically from the search space of the optimizer. In a distributed environment, there are two major cost components in parallel join strategies – data partitioning costs, and the costs directly associated with the join operation, such as CPU and local I/O costs of the intermediate results.

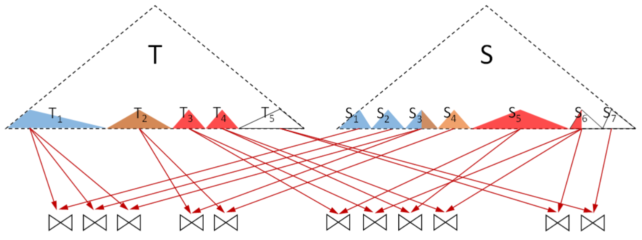


Figure 8: Partial-cross join graph.

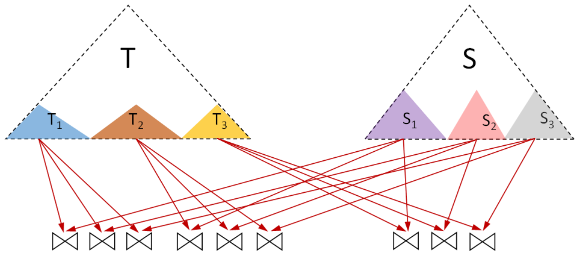


Figure 9: Complete bipartite join graph.

The trade-offs between symmetric (pair-wise) join graphs and full-cross join graphs are offsets of the data partitioning costs by the reduction of the join costs. In most cases, the reduction in join costs are much higher than the incurred costs of data partitioning. However, with the presence of interesting properties, it may allow further reductions of the join costs without the incurred data partitioning costs. The core idea is graph pruning of the *complete bipartite graph* by leveraging the interesting properties.

In the example, both table T and S are range partitioned, and suppose that $\mathsf{T_P} \sqsubseteq \mathsf{S_P}$ and $\mathsf{S_P} \subseteq \mathsf{S_J}$ (which implies $\mathsf{T_P} \subseteq \mathsf{T_J}$). Figure 8 shows the range boundaries of $\mathsf{RPART(T_P)}$ on both T and S in different color codes. Since $\mathsf{T_P} \sqsubseteq \mathsf{S_P}$, $\mathsf{RPART(S_P)}$ will have finer partition boundaries, resulting in some boundaries of $\mathsf{RPART(T_P)}$ reside inside physical partitions of $\mathsf{RPART(S_P)}$. Furthermore, since both $\mathsf{T_P} \subseteq \mathsf{T_J}$ and $\mathsf{S_P} \subseteq \mathsf{S_J}$, we can infer that, for example, the join of $\mathsf{T_2}$ and $\mathsf{S_5}$ will yield empty results. Thus, we can prune the connection between $\mathsf{T_2}$ and $\mathsf{S_5}$ from the full-cross graph. After pruning all unnecessary connections between $\mathsf{T}_i(i = 1 \ldots 5)$ and $\mathsf{S}_j(j = 1 \ldots 7)$, the resulting join graph is shown as Figure 8. Whether or not we should choose this execution plan over the conventional pair-wise join graph is a cost-based decision.

An important factor that affects the costs is the *join selectivity*. A complete bipartite join graph, shown in Figure 9, incurs a potentially very large data transfer and total costs, even though it does not incur data partitioning costs. Nevertheless, in rare but plausible conditions, it still provides a favorable alternative over the state-of-the-art pair-wise hash-based or sort-based join strategy. For instance, if the inner table (S) can provide interesting indexes to evaluate the join predicate efficiently and the join selectivity is very low, then both data transfer costs and the total join costs will remain low, since indexed loop join can leverage the indexing structures to push join operations to data. The lookup will only incur data transfer costs for rows that match the join predicate. In addition, with batched lookups, we can keep a good amortization of individual network round-trip costs.

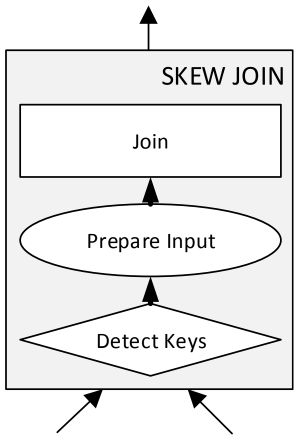


Figure 10: Three stages of `SkewJoin` transformation.

In this section, we discussed how join graph topologies enrich the solution space of join processing in MapReduce environment. In the following section, we will discuss how SCOPE applies different join graphs in tackling the join skew problems.

4. HANDLING JOIN SKEW

As illustrated in Section 1, handling data skew is a crucial challenge in large-scale distributed applications. We next present a class of logical transformations called `SkewJoin`. A `SkewJoin` transformation mitigates the impact of data skew in a distributed join operation. We present three concrete `SkewJoin` transformations and discuss their advantages and disadvantages.

4.1 Overview

The key idea of the `SkewJoin` is simple: separate high-frequency join key values and process them differently from low-frequency key values. That is, use different partitioning scheme and join graph topology for high-frequency key values. At a high-level, the `SkewJoin` transformation decomposes a join into three steps (see Figure 10):

1. **Detect high-frequency join key values.** The first step is identifying high-frequency (*i.e.*, *skewed*) values in the join key. If there is a mechanism to detect data skew in join key columns (*e.g.*, histograms, or approximate counts), this step can directly leverage the mechanism. If not, we generate a simple aggregation query to collect high frequency values (Section 4.4).

2. **Prepare join inputs.** Once the key values that require special handling are identified, the `SkewJoin` strategy prepares the join inputs by *splitting and/or repartitioning* the input tables accordingly. The low frequency values (*i.e.*, non-skewed values) will be prepared for an ordinary parallel join (*e.g.*, hash partitioned by join key with pair-wise join graph), while the high frequency values (*i.e.*, skewed values) will be prepared differently. We assume that the query execution engine supports reading intermediate data multiple times as well as repartitioning intermediate result. We present three alternatives that have different join graph topologies,

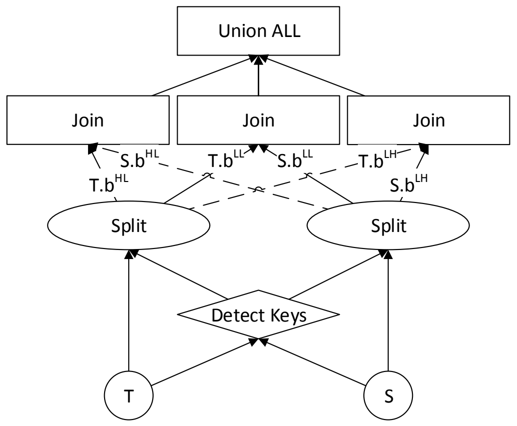


Figure 11: `B-SkewJoin` (*a.k.a.* Partial Replication Partial Redistribution).

different trade-offs between performance and robustness, and different capabilities in supporting join types and data skew scenarios (Section 4.2).

3. **Perform join proper.** `SkewJoin` strategies does not require a special join algorithm implementation. Any join algorithm implementation available in the query execution engine (*e.g.*, hash join, sort-merge join) can be used. However, certain join types require careful handling (Section 4.3).

In the rest of section, we use joining $\mathsf{T}(a, b)$ and $\mathsf{S}(b, c)$ on b as a running example. We also use b^{LL}, b^{HL}, b^{LH} to represent sets of join key values that are (i) not skewed in T and S, (ii) skewed in T but not skewed in S, and (iii) not skewed in T but skewed in S, respectively. If a key value has high-frequency in both tables, the higher frequency side takes precedence. We denote $\mathsf{T}.b^x$ as $\mathsf{T}.b^x = \{t | t \in \mathsf{T} \wedge t.b \in b^x\}$. For example, $\mathsf{S}.b^{LH}$ represent tuples in S that their b values are not skewed in T but skewed in S. For simplicity, we first restrict the discussion to inner join, then extend the approach to cover outer joins in Section 4.3.

4.2 SkewJoin Transformations

We describe the three alternative transformations to handle join skew, and discuss the benefits and drawbacks of each approach.

4.2.1 B-SkewJoin

`B-SkewJoin` (or Broadcast SkewJoin) is perhaps the most intuitive way to workaround join skew. We have observed many users who had a severe data skew problem had manually implemented this strategy by handcrafting their scripts before we implement SkewJoin in the system. `B-SkewJoin` is also known as *Partial Replication Partial Redistribution* (PRPD) [23]. `B-SkewJoin` mitigates data skew by *split*ting join input and using *broadcast join*. Suppose running $\mathsf{T} \bowtie_b \mathsf{S}$, $\mathsf{T}.b$ has a skewed distribution, and T is partitioned on a. By broadcasting S, join product skew is spread across the partitions as long as partitioning on $\mathsf{T}.a$ keeps the distribution of $\mathsf{T}.b$ relatively uniform. Splitting the input based on *skewed* values keeps the size of broadcasted rows small thus makes `B-SkewJoin` applicable to larger data.

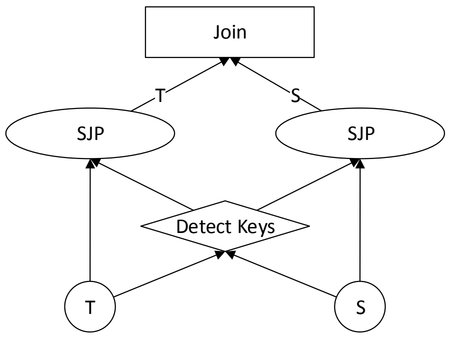


Figure 12: `F-SkewJoin`.

Figure 11 illustrates the graph topology of `B-SkewJoin` join at a high-level. Each join input is split into three disjoint subsets according to key distribution and processed by three distinct parallel joins. The non-skewed side (*i.e.*, $\mathsf{T}.b^{LH}$ and $\mathsf{S}.b^{HL}$) will be broadcasted (dashed lines) to the other side. The non-skewed input ($\mathsf{T}.b^{LL}$ and $\mathsf{S}.b^{LL}$) are processed by a pair-wise parallel join implementation. The results of these three sub-joins are put together to get the final result. `B-SkewJoin` is simple yet potentially the most efficient strategy for two reasons. First, this strategy does not require repartitioning input data if the input tables deliver interesting properties. Second, the result of join retains the data partitioning property of the non-broadcasted side thus can avoid repartitioning steps required by subsequent operators.

`B-SkewJoin` trades off reliability of execution time for performance because the overall performance heavily relies on the initial data partitioning of the non-broadcasted side. For example, suppose T is already partitioned on $\mathsf{T}.b$. In this scenario, broadcasting $\mathsf{S}.b^{HL}$ is not different from an ordinary pair-wise join because each high-frequency join key value is processed by a single node. Thus, `B-SkewJoin` should be used carefully by considering the input data partitioning scheme.

4.2.2 F-SkewJoin

The second alternative `F-SkewJoin` (or Full SkewJoin) is at the other extreme of trade-off between performance and reliability. Instead of avoiding repartitioning as much as possible, `F-SkewJoin` always fully repartitions the input data. Fully repartitioning data imposes greater overhead than `B-SkewJoin`. However, it results in better control over balancing join product skew by fine-tuning the join graph topology. Additionally, the overhead of repartitioning is more predictable than estimating the resulting join product skew from the initial data partitioning.

Specifically, `F-SkewJoin` repartitions the inputs as follows:

- Rows with **high-frequency values** in the table are *round-robin* partitioned to downstream join operators (*i.e.*, distribute without duplication).

- Rows with **high-frequency values** in the other table are replicated to downstream join operators (*i.e.*, distribute with duplication).

- Rows with **low-frequency values** in both tables are hash partitioned on the join key.

```
/* Called per key in L like a left outer join */
LeftSkewJoinPartitioner (L, R, o)
 // L: left input rows.
 // R: high-frequency key value from detection
 //  schema: (key, source, DOP)
 // o: output row. schema: (schema of L, partition)
  r = R.First()
  if (r is null)
     // Hash partition
     l.CopyTo(o)
     o.partition = Hash(r.key) % r.DOP
     yield return o
  else if (r.source is LEFT)
     // Round-robin partition
     i = 0
     foreach (l in L)
        l.CopyTo(o)
        o.partition = ++i % r.DOP
        yield return o
  else
     // Replicate
     foreach (l in L)
        l.CopyTo(o)
        for (i = 0; i < r.DOP; ++i)
           o.partition = i
           yield return o
```

Figure 13: Left skew join partitioner as a user-defined combiner.

In this way, F-SkewJoin effectively rebalances the non-broadcasted side of B-SkewJoin across multiple join operators. Each join operator processes both low frequency key values and high frequency key values. F-SkewJoin can vary parallelism of join operator to distribute the high frequency values if the information is available (*e.g.*, collected by a simple aggregation in the detection step as discussed in Section 4.4). Note that if F-SkewJoin always use full parallelism for each high-frequency key value, it behaves similar to B-SkewJoin.

To implement F-SkewJoin, a query execution engine needs a new partitioning operator, called Skew Join Partitioner, which repartitions and/or replicate the input table with respect to the high-frequency key information. If the system supports a user-defined operator (UDO), this Skew Join Partitioner can be implemented as a UDO. For example in SCOPE, we implemented Skew Join Partitioner as a user-defined *combiner* that implements a custom binary operation [24]. The combiner interface lets users enumerate and process all rows from both inputs bearing the same join key value. Figure 13 shows an example Skew Join Partitioner implemented as a SCOPE combiner UDO that prepares the left input according to high-frequency value detection results (*i.e.*, taking the left input table and high-frequency key values with their frequencies).

Figure 12 illustrates F-SkewJoin at a high-level. Skew Join Partitioner (SJP) prepares each join input by replicating (T.b^{LH} and S.b^{HL}), partitioning in round-robin (T.b^{HL} and S.b^{LH}), and hash-partitioning (T.b^{LL} and S.b^{LL}) depending on the detection result. The prepared input is processed by a single parallel join thus does not require union.

Although F-SkewJoin is more versatile and reliable than B-SkewJoin, always repartitioning input data may incur high overhead for very large dataset. Additionally, if the join operator is part of a more complex pipeline, subsequent operators

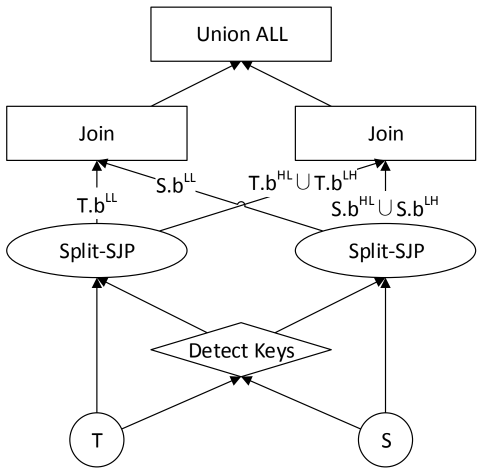


Figure 14: H-SkewJoin.

may require an extra repartitioning step because Skew Join Partitioner breaks the existing input data partitioning.

4.2.3 H-SkewJoin

The last alternative H-SkewJoin (Hybrid SkewJoin) is a hybrid of B-SkewJoin and F-SkewJoin. It strikes balance between performance and reliability. This strategy splits the join inputs as in B-SkewJoin but uses Skew Join Partitioner to repartition the skewed side as in F-SkewJoin.

By splitting the input, H-SkewJoin can retain the original data partitioning on the non-skewed side (*i.e.*, $*.b^{LL}$) but loses it in the skewed side. If majority of join key values are distinct, H-SkewJoin can avoid expensive repartitioning compared to F-SkewJoin. However, splitting the dataflow into two parts might result in more operators to schedule at runtime, and such extra overheads could counter the benefits of preserving the original data partitioning.

Figure 14 illustrates H-SkewJoin join graph topology. The join inputs are first split as in B-SkewJoin ($*.b^{LL}$ and the rest), then tuples bearing high-frequency values are processed by Skew Join Partitioner. Two parallel joins process the data then we union partial results to obtain the final answer.

Summary. In this section, we reviewed three SkewJoin transformations. Each transformation has a unique trade-off between performance and reliability. We summarize the three transformations in Table 3.

4.3 Handling different Join Types

The actual join operation in SkewJoin can be done using any join algorithm implementation. However, data skew in certain outer join scenarios requires special attention for correctness. For example, as discussed in Section 3, broadcasting the left input to the right is not correct in a left outer join because those broadcasted left rows will produce wrong null values when they do not join with the tuples from the right. Thus, B-SkewJoin can not handle data skew in the inner table of an outer join.

F-SkewJoin and H-SkewJoin, in turn, can support this scenario by ensuring that at least one tuple is distributed to each downstream partition when repartitioning skewed key values from inner table (*e.g.*, S.b^{LH}) in round robin fashion. The property is satisfied if the frequency of a value is high

		`B-SkewJoin`		`F-SkewJoin`		`H-SkewJoin`	
	Source of data skew	Left	Right	Left	Right	Left	Right
Applicability	Inner	√	√	√	√	√	√
	Left Outer	×	√	√	√	√	√
	Right Outer	√	×	√	√	√	√
	Full Outer	×	×	√	√	√	√
Partition Requirements	Repartition low-frequency keys	if necessary		Yes		if necessary	
	Repartition high-frequency keys	No		Yes		Yes	
	Require SJP	×		√		√	
Join Graph Topology	High-Frequency Key Source	None		Round-Robin		Round-Robin	
	High-Frequency Key Other	Broadcast		Partial Replication		Partial Replication	
	Join Low-Frequency Keys	Separate		Not Separate		Separate	
Derived Interesting Properties	Preserve Partitioning	Yes		No		Partial	
	Preserve Sort	Yes		Yes		Yes	
Cost	Overhead	Low		High		Moderate	
	Robustness	Low		High		High	
	# of Scheduled Operators	High		Low		Moderate	

Table 3: Summary of `SkewJoin` transformations.

enough. Specifically, whenever the high-frequency values are repeated at least roughly $|D| * | \bowtie |/2$ where $|D|$ is the number of input partitions (*e.g.*, $|\mathsf{T}|$, $|\mathsf{S}|$), and $| \bowtie |$ is the number of partitions of the join operator.

If that is not the case, `Skew Join Partitioner` can enforce this property by generating witness tuples for each key value partitioned in round-robin so that each join operator sees at least one original or witness tuple. The witness tuple prevents an outer join operator from generating null values but results produced by a witnessed tuple must be excluded from the final result. This can be implemented as an extra filter after the join or a modified outer join operator variant that natively understands witness tuple semantics.

4.4 Detecting High Frequency Key Values

The three `SkewJoin` alternatives discussed in previous sections rely on the information about high frequency key values. In this section, we answer the following questions through cost analysis. First, what is the threshold of high-frequency values? And second, how many partitions need to be used? We assume fully pipelined execution for simplicity and different execution model may require adjustment of the results. We assume joining $m \times n$ $(m > n)$ tuples using K processors where both inputs are already partitioned on join key. This is the worst scenario in `SkewJoin` because both replication and redistribution process the entire data for each join key value.

Threshold for high-frequency key values

Given frequency of a value, when is it beneficial to replicate? Let c_j, c_r, c_x be cost of join per output tuple, cost of replicating/broadcasting one tuple to other processor, and cost of transferring one tuple to other processor respectively. `SkewJoin` is beneficial when running the join in a single node takes longer than running it in multiple nodes. When running in multiple nodes, n tuples will be replicated K times and each processor produces $\frac{1}{K}$ output tuples. m tuples will be evenly distributed to K processors in a round-robin fashion. Then, `SkewJoin` is beneficial when the latency of a serial join is greater than that of `SkewJoin`:

$$m \cdot n \cdot c_j > \max(m \cdot c_x, n \cdot K \cdot c_r) + \frac{m}{K} \cdot n \cdot c_j$$

Depending on the cost of round-robin distribution and replication, we get following inequalities:

$$m > \frac{K^2}{K-1}\frac{c_r}{c_j} > K \cdot \frac{c_r}{c_j} \qquad \text{if } m \cdot c_x \leq n \cdot K \cdot c_r$$

$$n > \frac{K}{K-1}\frac{c_x}{c_j} > \frac{c_x}{c_j} \qquad \text{if } m \cdot c_x > n \cdot K \cdot c_r$$

Thus, we can apply `SkewJoin` for keys of which frequencies are above the thresholds. Although the two inequalities need to be applied depending on relative costs of redistribution and replication, we found that threshold on m (*i.e.*, higher frequency between the two) is sufficient in practice. The coefficients can be determined as system-wide constants and K is often known in advance from the system configuration. Therefore, the threshold can remain intact per data change, which is not possible with a relative threshold (*e.g.*, fraction of tuples). Having an absolute threshold has another advantage with heavy-tailed distribution especially where there are hundreds of thousands of keys that are better to join in parallel at the cost of the overhead.

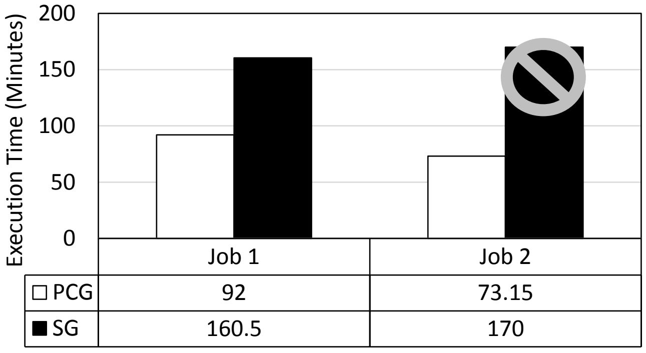
Parallelism for each high-frequency key value

The distribution of join keys might have one or two values with extreme frequencies but it is also possible that there are hundreds or thousands of values with relatively high frequencies. Clearly, if frequency is not extreme, it may not worth to use all K processors to balance the workload. By rearranging the threshold inequality, the desired parallelism k with respect to the frequency m is calculated as:

$$k \leq \left\lfloor \frac{c_j}{c_r} \cdot m \right\rfloor$$

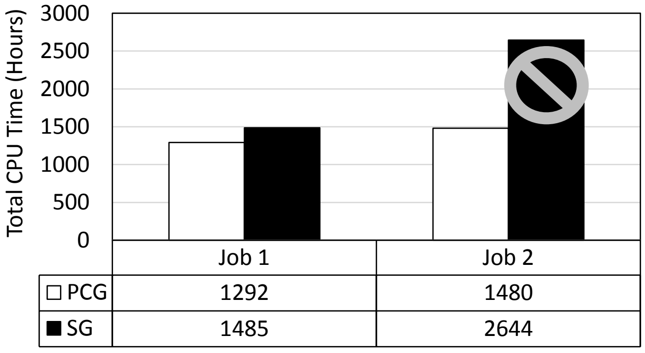
$$k = \min(\left\lfloor \frac{c_j}{c_r} \cdot m \right\rfloor, K)$$

If we adapt the parallelism per frequency, then the threshold inequality should use the minimum desired parallelism (*e.g.*, 2) for K.



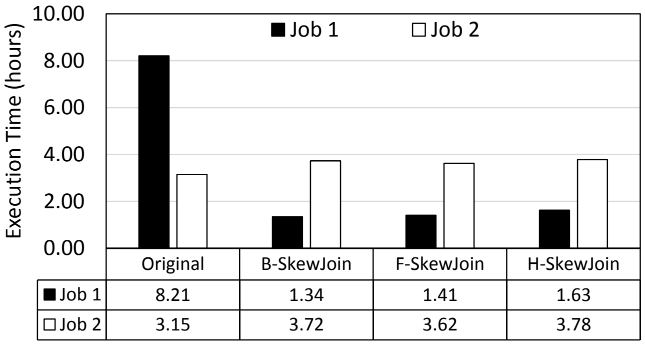
	Job 1	Job 2
☐ PCG	92	73.15
■ SG	160.5	170

Figure 15: Job execution latency.



	Job 1	Job 2
☐ PCG	1292	1480
■ SG	1485	2644

Figure 16: Total CPU time aggregated across all parallel execution.



	Original	B-SkewJoin	F-SkewJoin	H-SkewJoin
■ Job 1	8.21	1.34	1.41	1.63
☐ Job 2	3.15	3.72	3.62	3.78

Figure 17: Difference in execution time among SkewJoin alternatives in the presence of data skew.

5. EXPERIMENTAL EVALUATION

We implemented the different join strategies introduced in this work in the SCOPE system, which is deployed on production clusters consisting of tens of thousands of machines at MICROSOFT. SCOPE clusters execute tens of thousands of jobs daily, reading and writing tens of petabytes of data in total, and powering different online services. Our experiments were conducted on a cluster with machines with two six-core AMD Opteron processors running at 1.8GHz, 24 GB of RAM, and four 1TB SATA disks. All machines run Windows Server 2008 R2 Enterprise X64 Edition.

5.1 Join Graph Topologies

We first evaluate the performance of different join graph topologies. In particular, as stated in Section 3, asymmetric join graphs are less intuitive to outperform symmetric join graphs. As a result, traditional parallel database systems as well as recent big data processing platforms often overlook them from the solution space. When interesting properties are present, asymmetric join graphs become attractive, due to the facts that (i) there are no incurred repartitioning costs, and (ii) the complete bipartite graph can be reduced to partial-cross join graph based on interesting properties.

We picked two jobs, each with a single two-table join with different join selectivity values. The bigger table, T, is approximately 500 TB in size, while the other smaller table, S, is approximately 30 TB in size. Table T is range partitioned on columns sequence $\langle \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d} \rangle$ into m partitions, and table S is range partitioned on columns $\langle \mathsf{a}, \mathsf{b} \rangle$ into n partitions[7]. The join predicate is $\mathsf{T.a} = \mathsf{S.a}$ AND $\mathsf{T.b} = \mathsf{S.b}$.

We measure two data points - job execution latency (shown in Figure 15) and total CPU time aggregated across all parallel execution (shown in Figure 16). Job 1 has a selection predicate that results in very low join selectivity ($< 5\%$), and Job 2 has very high join selectivity ($> 90\%$). The two parallel join strategies in the comparisons are joins with partial-cross join graph (PCG) such as those in Figure 8, and joins with (pair-wise) symmetric join graph (SG). We can see that with the interesting partitioning properties, PCG outperforms SG in both cases due to the huge costs associated with repartitioning totally 530 TB of data. Interestingly, the reason why the total CPU time of PCG is only slightly lower than that of SG for Job 1 is that the execution with PCG will read part of the underlying data twice due to the overlapping partition boundaries. In addition, since the highly selective predicate is pushed before join, the total data volume decreases a lot, thus reduce the differences between these two strategies. However, the big improvements in latency are due to the saving of eliminating repartitioning in the PCG run. As for Job 2, the SG run eventually failed (killed by the system due to policies to prevent run-away jobs) after 170 minutes into the job execution due to the huge amount of data repartitioning. On the other hand, the PCG run completed within 74 minutes.

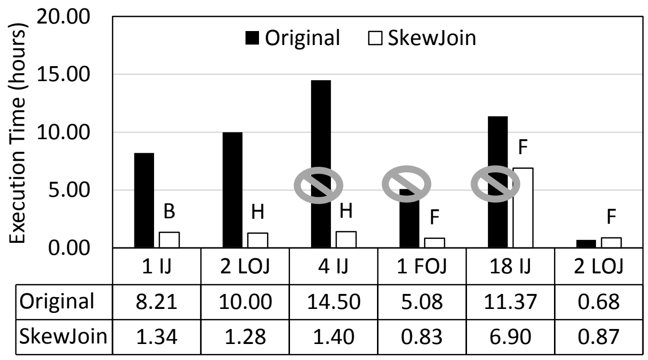
[7]Without loss of generality, the sorting order use in the example is ascending.

The join algorithm in the experiments is the sort-merge join. For different join algorithms, the break-even point between PCG and SG will be different. Nevertheless, with the above experiments we demonstrate that there are interesting parallel join strategies with asymmetric join graphs that can be explored by the system to further reduce the join processing time.

5.2 SkewJoins

We next evaluate the characteristics of the various SkewJoin alternatives and discuss their relative performance in real workloads on production clusters.

To compare the relative performance of the various SkewJoin alternatives, we executed two representative jobs from the production workload forcing each SkewJoin variant. Figure 17 shows the total elapsed time of each approach. In Job 1, there is a single inner join that has severe data skew in the join key (the highest frequency is in the order of millions). In Job 2, there are two left outer joins in which the join key has a mild data skew (the highest frequency is in the order of thousands). As shown in the figure, there is no SkewJoin alternative that consistently performs the best. In



	1 IJ	2 LOJ	4 IJ	1 FOJ	18 IJ	2 LOJ
Original	8.21	10.00	14.50	5.08	11.37	0.68
SkewJoin	1.34	1.28	1.40	0.83	6.90	0.87

Figure 18: Difference in execution time with and without SkewJoin in 6 distinct jobs. IJ, LOJ, and FOJ are Inner Join, Left Outer Join, and Full Outer Join respectively. B, F, and H represent B-SkewJoin, F-SkewJoin, and H-SkewJoin.

Job 1, `B-SkewJoin` yields the best runtime while `F-SkewJoin` is the best in Job 2 among the three SkewJoin transformations. `H-SkewJoin` performed the worst in the two jobs but it can outperform `F-SkewJoin` where interesting properties are desired by the following operations.

All three SkewJoin alternatives come with their own pros and cons. The gains depend on the severity of data skew in join keys. For example, with a severe data skew in Job 1, all three alternatives significantly improved the execution time by more than factor of 5. On the other hand, in Job 2, all three alternatives run slightly slower than the original job where the data skew problem is not severe enough to compensate the overhead of SkewJoin. The overhead is proportional to the intermediate data size and trading the performance over reliable runtime at the cost of overhead (*e.g.*, from 14% to 20% in Job 2). Such overhead may or may not be acceptable depending on user requirements and one of the criteria in choosing the transformation.

Figure 18 compares elapsed time of six production jobs in production clusters with and without SkewJoin strategies. Original jobs use conventional pair-wise parallel join implementation and SkewJoin jobs use SkewJoins for some of joins in the original jobs. For example, 2 LOJ with H label means that two of the left outer joins in the job use H-SkewJoin. Some original jobs (Job 3, 4, 5) did not run to finish because some vertices ran for too long and were killed by the system or the user. Such jobs are marked with grey cross signs. Overall, the degree of improvement varies per data distribution, per job but we generally observe at least factor of 4 to 10 improvements in terms of latency. If the data skew is not significant enough, SkewJoin runs slightly slower than the original job as shown in the 2 LOJ with F-SkewJoin case in Figure 18.

Summary: In this section we showed that `SkewJoin` strategies effectively mitigate different data skew scenarios in inner- and outer- joins. The benefits of balancing join workload depend on the severity of data skew in join key. The three alternatives of SkewJoin strategy have their own pros and cons and there is no single alternative that outperforms the others in all occasions.

6. RELATED WORK

Join processing has been extensively studied in the database community for several decades. Graefe gave a comprehensive survey on query processing for large databases, including join types, join orders and join algorithms [7]. More recently, new research emerged regarding improvements on some specific aspects of join processing in MapReduce environments, such as multi-way joins [1], outer join with skew [22], and broadcast-joins [16]. In this paper, we categorize new aspects of parallel join processing in MapReduce environment into a new dimension of the solution space – join graph topology. To the best of our knowledge, there is no prior work proposing such a taxonomy. Instead, there is recent work that tackle individual factors of the join graph topology – in particular regarding partition schemes, data exchange operators and distribution polices [3]. A formal taxonomy allows us to systematically discover new improvements and identify missing optimization opportunities.

The problem of skew in parallel databases has been extensively studied by many research groups, especially in the context of the join operator [5, 9, 10, 12, 13, 17, 20, 21]. Specifically, these references range from exploiting low-level system support (*e.g.*, network switches or operating system features) [12, 17], to optimizing data distribution among processors [9, 10], to scheduling partitions in an optimal order [20, 21], to improving a specific join algorithm [13].

DeWitt *et al.* proposed a practical approach to handle skew in a parallel join [5]. They considered combinations of range partitioning (with subset-replication, weighting), load scheduling (*e.g.*, round-robin, longest processing time (LPT)), and virtual processors (*i.e.*, create many logical partitions and schedule multiple partitions per physical processor). For range partitioning, DeWitt *et al.* proposed an efficient sampling technique to estimate the distribution of the join key value. For LPT scheduling, they used a simple cost model to estimate the cost of each partition. As it is relying on range partitioning, this technique can handle redistribution skew but not the join product skew very well. The technique is easy to implement yet effective in practice. At a high-level, `F-SkewJoin` is a hash variant of virtual processor partitioning with round-robin strategy in DeWitt *et al.* with improvements on handling all data skew scenarios as well as outer joins. Pig, a declarative layer of Hadoop, implements DeWitt *et al.* (virtual processors with round-robin in range partitioning) and extends the technique to handle outer join [6, 18]. However, it can handle data skew in one input of the inner join and has the same limitations as `B-SkewJoin` in outer join.

Xu *et al.* proposed a parallel join approach, partial redistribution partial duplication (PRPD) to handle data skew [23]. The PRPD approach first splits the rows of joining relation into three disjoint groups that are handled differently (redistributed, duplicated, or kept local). The rows having skewed values are kept local, the rows bearing skewed values of joining attributes in the other relation are duplicated, and the remaining rows are redistributed as in an ordinary parallel hash join. The final join is computed by union of three joins (two replicated-joins for skewed join values of each relation, and an ordinary parallel join for non-skewed values). In Section 4, we captured PRPD as `B-SkewJoin` transformation in the SkewJoin framework and discussed its pros and cons. Xu *et al.* also proposed a parallel outer join algorithm OJSO (Outer Join Skew Optimization) in the presence of data skew [22]. OJSO handles data skew in *a chain of outer joins* by separating result of each outer join into nulls (*i.e.*, joined) and non-nulls (*i.e.*, not joined), and only processing non-nulls in following outer join. OJSO can be orthogonally applied to the chain of outer joins with SkewJoins.

Hive, another popular declarative layer of Hadoop MapReduce, also implements two complementary strategies to handle data skew in join [8, 14]. The first strategy is a run time strategy which the join spills high-frequency keys to HDFS then process them in follow-up conditional map-side joins (*i.e.*, broadcast join). As of Hive 0.12, this approach only supports an inner join. The second strategy is a compile time strategy similar to `B-SkewJoin`: it splits the input into the skewed and the non-skewed, then uses map-side join to handle the data skew. This approach has the same limitation and applicability as that of Pig.

7. CONCLUSIONS

Massive data analysis in cloud-scale data centers plays a very important role in making critical business decisions. High-level scripting languages free users from understanding various system trade-offs and complexities, support a transparent abstraction of the underlying system, and provide the system great opportunities and challenges for query optimization. In this paper, we studied the very common join operation and identified challenges and opportunities in the distributed setting. Specifically, we introduced a generic taxonomy that results in new join strategies, and new query rewrites that deal with data skew joins robustly. Experiments on a large-scale SCOPE production system at Microsoft show that the proposed techniques systematically solve the challenges of data skew and generally improve query latency by a significant margin.

8. REFERENCES

[1] F. Afrati and J. Ullman. Optimizing multiway joins in a map-reduce environment. *IEEE Transactions on Knowledge and Data Engineering*, 23(9):1282–1298, 2011.

[2] Apache. Hadoop. http://hadoop.apache.org/.

[3] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in MapReduce. In *Proc. of the SIGMOD Conf.*, pages 975–986, 2010.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of OSDI Conference*, pages 10–10, 2004.

[5] D. DeWitt, J. Naughton, D. Schneider, and S. S. Seshadri. Practical skew handling in parallel joins. In *Proc. of the 18th VLDB Conf.*, pages 27–40, 1992.

[6] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proc. of the VLDB Endowment*, 2:1414–1425, August 2009.

[7] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–169, 1993.

[8] He Yongqiang. handle skewed keys for a join in a separate job. https://issues.apache.org/jira/browse/HIVE-964.

[9] K. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In *Proc. of the 17th VLDB Conf.*, pages 525–535, 1991.

[10] K. Hua, C. Lee, and C. Hua. Dynamic load balancing in multicomputer database systems using partition tuning. *IEEE TKDE*, 7(6):968–983, 1995.

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. of EuroSys Conference*, pages 59–72, 2007.

[12] M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash: a new, robust, parallel hash join method for data skew in the super database computer (sdc). In *Proc. of the 16th VLDB Conf.*, pages 210–221, 1990.

[13] W. Li, D. Gao, and R. Snodgrass. Skew handling techniques in sort-merge join. In *Proc. of the SIGMOD Conf.*, pages 169–180, 2002.

[14] Namit Jain. Skewed Join Optimization. https://issues.apache.org/jira/browse/HIVE-3086.

[15] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of SIGMOD Conference*, pages 1099–1110, 2008.

[16] S. Schelter, C. Boden, M. Schenck, A. Alexandrov, and V. Markl. Distributed matrix factorization with mapreduce using a series of broadcast-joins. In *Proc. of the 7th ACM conf. on Recommender Systems*, pages 281–284, 2013.

[17] A. Shatdal and J. F. Naughton. Using shared virtual memory for parallel join processing. In *Proc. of the SIGMOD Conf.*, pages 119–128, 1993.

[18] Sriranjan Manjunath. support for skewed outer join. https://issues.apache.org/jira/browse/PIG-1035.

[19] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive – a petabyte scale data warehouse using Hadoop. In *Proceedings of ICDE Conference*, pages 996–1005, 2010.

[20] J. L. Wolf, D. M. Dias, and P. S. Yu. An effective algorithm for parallelizing sort merge joins in the presence of data skew. In *Proceedings of the second international symposium on databases in parallel and distributed systems*, DPDS '90, pages 103–115, 1990.

[21] J. L. Wolf, D. M. Dias, P. S. Yu, and J. Turek. An effective algorithm for parallelizing hash joins in the presence of data skew. In *Proc. of the 7th ICDE Conf.*, pages 200–209, 1991.

[22] Y. Xu and P. Kostamaa. Efficient outer join data skew handling in parallel dbms. *Proc. of the VLDB Endowment*, 2(2):1390–1396, 2009.

[23] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen. Handling data skew in parallel joins in shared-nothing systems. In *Proc. of the SIGMOD Conf.*, pages 1043–1052, 2008.

[24] J. Zhou, N. Bruno, M.-C. Wu, P.-Å. Larson, R. Chaiken, and D. Shakib. SCOPE: Parallel databases meet mapreduce. *The VLDB Journal*, 21(5):611–636, 2012.

[25] J. Zhou, P.-Å. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *Proceedings of ICDE Conference*, pages 1060–1071, 2010.