Advanced Partitioning Techniques for Massively Distributed Computation

Jingren Zhou
Microsoft
jrzhou@microsoft.com

Nicolas Bruno
Microsoft
nicolasb@microsoft.com

Wei Lin
Microsoft
weilin@microsoft.com

ABSTRACT

An increasing number of companies rely on distributed data storage and processing over large clusters of commodity machines for critical business decisions. Although plain MapReduce systems provide several benefits, they carry certain limitations that impact developer productivity and optimization opportunities. Higher level programming languages plus conceptual data models have recently emerged to address such limitations. These languages offer a single machine programming abstraction and are able to perform sophisticated query optimization and apply efficient execution strategies. In massively distributed computation, data shuffling is typically the most expensive operation and can lead to serious performance bottlenecks if not done properly. An important optimization opportunity in this environment is that of judicious placement of repartitioning operators and choice of alternative implementations. In this paper we discuss advanced partitioning strategies, their implementation, and how they are integrated in the Microsoft SCOPE system. We show experimentally that our approach significantly improves performance for a large class of real-world jobs.

Categories and Subject Descriptors

H.2 [**Information Systems**]: Database Management

Keywords

Scope, Partitioning, Distributed Computation, Query Optimization

1. INTRODUCTION

An increasing number of companies rely on the results of massive data computation for critical business decisions. Such analysis is crucial in many ways, such as to improve service quality and support novel features, to detect changes in patterns over time, and to detect fraudulent activities. Usually the scale of the data volumes to be stored and processed is so large that traditional, centralized database system solutions are no longer practical or even viable.

For that reason, several companies have developed distributed data storage and processing systems on large clusters of thousands of shared-nothing commodity servers. Examples of such initiatives include Google's MapReduce [5], Hadoop [1] from the open-source community and used at Yahoo, and Cosmos/Dryad [3, 12] at Microsoft. In the MapReduce approach, developers provide map and reduce functions in procedural languages like C++, which perform data transformation and aggregation. The underlying runtime system achieves parallelism by partitioning the data and processing each partition concurrently using multiple machines. This model scales reasonably well to massive data sets and has sophisticated mechanisms to achieve load-balancing, outlier detection, and recovery to failures, among others.

This approach, however, has its own set of limitations. Users are required to translate their application logic to the MapReduce model in order to achieve parallelism. For some applications this mapping is very unnatural. Users have to provide implementations for the map and reduce functions, even for simple operations like projection and selection. Such custom code is error-prone and hardly reusable. Moreover, for complex applications that require multiple MapReduce stages, there are often many valid evaluation strategies and execution orders. Having developers manually implement and combine multiple MapReduce functions is equivalent to asking them to specify physical execution plans directly in relational database systems, an approach that became obsolete with the introduction of the relational model over three decades ago. Moreover, optimizing complex, multi-step MapReduce jobs is difficult, since it is not usually possible to do complex reasoning over sequences of opaque, primitive MapReduce operations.

To address this problem, higher level programming languages plus conceptual data models were recently proposed, including Jaql [2], SCOPE [3, 24], Tenzing [4], Dremel [15], Pig [18], Hive [21], and DryadLINQ [23]. These languages offer a single machine programming abstraction and allow developers to focus on application logic, while providing systematic optimizations for the distributed computation. These optimizations take advantage of a full view of the application logic, and can perform sophisticated query optimization strategies that are simply not possible otherwise.

In massive distributed computation, data shuffling is typically the most expensive operation and can lead to serious performance bottlenecks if not done properly. A very important optimization opportunity in this type of environment is that of judicious placement of repartitioning operators and choice of different implementation techniques. Complex scripts and reasoning about properties of intermediate results opens the door to a rich class of optimization opportunities regarding data partitioning. It is crucial to avoid repartitioning unless absolutely necessary, and do so as efficiently as possible. Previous work focused on minimizing the number of partitioning operations while executing complex scripts [24]. However, independently on how much we optimize input scripts, there are still scenarios that require data shuffling. There are also different ways to perform data shuffling, and each one is preferable under different conditions. The optimal choice of shuffling operations depends on many factors, including data and code properties, and scalability requirements. It is valuable to integrate the reasoning with the query optimizer and systematically consider alternatives with the rest of the query.

In this paper we describe advanced partitioning techniques that leverage input data properties and optimize data movement across the network. As a result, we are able to greatly improve the shuffling efficiency, by either avoiding unnecessary repartitioning or partially repartitioning the input data set. The techniques are applicable both at compilation time and at runtime. We also design a novel partitioning strategy which indexes intermediate partitioned results and stores them in a single sequential file. This approach fundamentally solves the scalability challenge for data partitioning and efficiently supports an arbitrarily large number of partitions.

All the partitioning techniques are implemented into the Scope system at Microsoft, which is running in production over tens of thousands of machines. The query optimizer of Scope considers different alternatives in a single optimization framework and chooses the optimal solution in a cost-based fashion. Experiments show that the proposed techniques improve data shuffling efficiency by a few folds for real-world queries. Although this paper uses Scope as the underlying data and computation platform, the ideas are applicable to any distributed system that relies on a query optimizer and performs data shuffling.

The rest of the paper is structured as follows. In Section 2 we review the necessary technical background to support the remaining sections. In Section 3 we describe the implementation of partial repartitioning and its interaction with the query optimizer. In Section 4 we discuss a scalable index-based partitioning strategy. Section 5 reports an experimental evaluation of our strategies discussed on real-world data. Section 6 discusses related work, and Section 7 concludes the paper.

2. PRELIMINARIES

Scope [3, 24] (Structured Computations Optimized for Parallel Execution) is the distributed computation platform for Microsoft's online services targeted for large scale data analysis. It incorporates the best characteristics of both MapReduce and parallel database systems. The system runs over large clusters of tens of thousands of machines, executes tens of thousands of jobs daily, and it is on its way to become an exabyte store. We start with an overview of Scope, including the query language, query optimization, and different partitioning types. The reader can find more details on these concepts in the literature [3, 24, 12].

2.1 Query Language

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 deeply 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). This flexibility allows users to solve problems that cannot be easily expressed in SQL, while at the same time is able to perform sophisticated reasoning of scripts.

Figure 1(a) shows a very simple Scope script that counts the different 4-grams of a given single-column string data set. In the figure, `NGramProcessor` is a `C#` user defined operator that outputs, for each input row, all its n-grams ($n = 4$ in the example). Conceptually, the intermediate output of the processor is a regular rowset that is processed by the SQL-like main 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 based on Cascades framework [9], and 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 Scope optimizer extends Cascades by fully integrating parallel plan optimization and performing more complex structural data property reasoning, such as partitioning and sorting of intermediate results.

The optimizer returns an execution plan that specifies the steps that are required to efficiently execute the script. Figure 1(c) shows the output from the optimizer, which defines specific implementations for each operation (e.g., stream-based aggregation), data partitioning operations (e.g., the repartition operator), and additional implementation details (e.g., the initial sort after the processor, and the unfolding of the aggregate into a local/global pair).

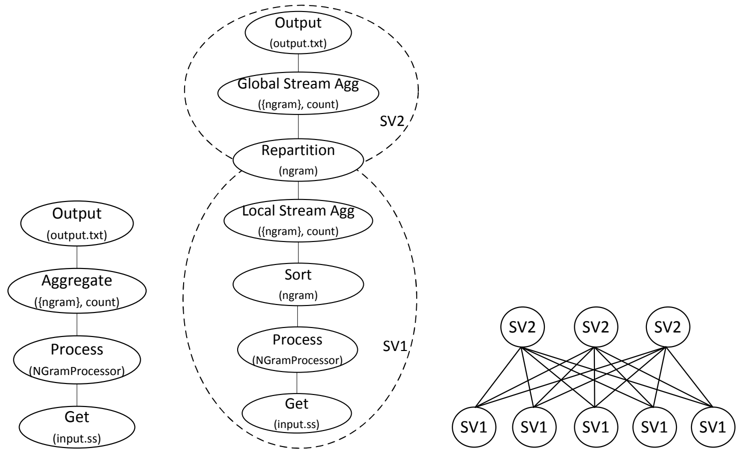
Finally, code generation produces the final algebra (which details the units of execution and data dependencies among them) and the assemblies that contain both user defined code and runtime implementation of the operators in the execution plan. Figure 1(c) shows dotted lines for the two units of execution corresponding to the input script. This package is then sent to the cluster, where it is actually executed. Users can monitor the progress of running jobs, and there are management utilities to submit, queue, and prioritize scripts in the shared cluster.

2.3 Data Partitioning

A key feature of distributed query processing is based on partitioning data into smaller subsets and processing partitions in parallel on multiple machines. This requires opera-

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

OUTPUT TO "output.txt";
```

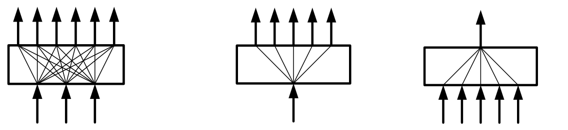


(a) SCOPE script (b) Input tree (c) Output tree (d) Scheduled graph

Figure 1: Definition, Compilation, and Scheduling of a Simple SCOPE Script

tors for splitting a single input into smaller partitions, merging multiple partitions into a single output, and repartitioning an already partitioned input into a new set of partitions. This can done by a single operator, the *data exchange* operator, that repartitions data from n inputs to m outputs [8]. After an exchange operator, the data is partitioned into m subsets that are then processed independently and in parallel using standard operators, until the data flows into the next exchange operator.

Exchange is implemented by one or two physical operators: a partition operator and/or a merge operator. Each partition operator simply partitions its input while each merge operator collects and merges the partial results that belong to its result partition. Suppose we want to repartition n input partitions, each one on a different machine, into m output partitions on a different set of machines. The processing is done by n partition operators, one on each input machine, which read its input and split it onto m local partitions, and m merge operators, one on each output machine, which collect the data for their partition from the n corresponding local partitions. Figure 1(d) shows five partition operators at the end of each SV1 vertex instance split their input into three partitions each, and three merge operators at the beginning of each SV2 vertex instance merge the five pieces of a corresponding partition.



(a) Full Repartitioning (b) Initial Split (c) Full Merge

Figure 2: Different Types of Data Exchange

2.3.1 Exchange Topology

Figure 2 shows the main classes of exchange operators. *Full Repartitioning* (Figure 2(a)) consumes n input partitions and produces m output partitions, partitioned in a different way. Every input partition contributes to every output partition, resulting in $n \cdot m$ connections. In the example of Figure 1(c), the plan uses full repartitioning on the `nGram` column in SV1, so that the input, which is not partitioned by `nGram`, can be aggregated correctly by SV2. *Initial Split* (Figure 2(b)) is a special case of full partitioning where there is a single input stream that is partitioned into m streams, without merge operators. Finally, *Full Merge* (Figure 2(c)) is a special case of full partitioning when there is a single output stream, merged from n input streams without partition operators. In this paper we explore alternative implementations of exchange operators that are based on additional topologies.

2.3.2 Partitioning Schemes

An instance of a partition operator takes one input stream and generates multiple output streams. It consumes one row at a time and writes the row to the output stream selected by a partitioning function applied to the row in a FIFO manner (so that the order of two rows r_1 and r_2 in the input stream is preserved if they are assigned to the same partition). There are several different types of partitioning schemes. *Hash Partitioning* applies a hash function to the partitioning columns to generate the partition number to which the row is output. *Range Partitioning* divides the domain of the partitioning columns into a set of disjoint ranges, as many as the desired number of partitions. A row is assigned to the partition determined by the value of its partitioning columns, producing ordered partitions. Other *non-deterministic* partitioning schemes, in which the data content of a row does not affect which partition the row is assigned to, include round-robin and random. The example in Figure 1(c) uses hash partitioning.

2.3.3 Merging Schemes

An instance of a merge operator combines data from multiple input streams into a single output stream. Depending on whether the input streams are sorted individually and how rows from different input streams are ordered, we have several types of merge operations. *Random Merge* randomly pulls rows from different input streams and merges them into a single output stream, so the ordering of rows from the *same* input stream is preserved. *Sort Merge* takes a list of sort columns as a parameter and a set of input streams sorted on the same columns. The input streams are merged together into a single sorted output stream. *Concat Merge* concatenates multiple input streams into a single output stream. It consumes one input stream at a time and outputs its rows *in order* to the output stream. That is, it maintains the row order within an input stream but it does not guarantee the order in which the input streams are consumed. Finally, *Sort-Concat Merge* takes a list of sort columns as a parameter. First, it picks one row (usually the first one) from each input stream, sorts them on the values on the sort columns, and uses the row order to decide the order in which to concatenate the input streams. This is useful for merging range-partitioned inputs into a fully ordered output. In the example of Figure 1(c), the different partitions are ordered by `nGram` due to the sort operator, so we use the *Sort Merge* variant to ensure the right row ordering when consumed by the global stream aggregate operator.

2.4 Job Scheduling

Scope relies on Dryad [12] to schedule the compiled script inside the cluster. The execution of a script can be modeled as a graph, where each vertex represents an instance of a computation unit, and each edge corresponds to data flow between vertices [1]. Figure 1(d) shows the execution graph for the sample script, assuming that the input data is laid out in five machines and the optimizer determines that data would be better aggregated into three partitions.

Vertex scheduling is rather sophisticated and takes different factors into consideration when deciding which vertices to run next, and on which machine to place such vertices (e.g., data locality, average execution time and memory consumption). Additionally, the nature of the graph imposes some natural scheduling constraints. A vertex can only start when all its inputs are already finished processing. For instance, in Figure 1(d), any SV2 vertex can only begin after all SV1 vertices finish. This is required not only to avoid producers and consumers running concurrently, but also because we need to simultaneously *Sort-Merge* SV2's inputs.

2.4.1 Aggregation Trees

Vertex scheduling attempts to minimize the overall job latency. One important aspect to achieve this goal is to reduce recovery costs due to failures. Consider a merge operator that consumes data from a very large number of partition vertices (i.e., suppose in Figure 1(d) that there are a thousand SV1 partition vertices connecting to each one of the three SV2 vertices). The chance of a random failure in SV2 while reading SV1 outputs increases with the number of input connections. Moreover, any such failure causes the whole SV2 vertex instance to restart, wasting partial work. To alleviate this problem, aggregates are typically done using aggregation trees, which can be seen as checkpoints during aggregation. When the number of input connections to an aggregate vertex is beyond a certain limit, we introduce partial aggregate vertices that operate over fragments of the input partition vertices. This works well for queries where merge operations interact with algebraic partial aggregation, which reduces the input size and can be applied multiple times at different levels without changing query correctness. We can then aggregate the inputs within the same rack before sending them out, reducing the overall network traffic among racks. As an example, if the threshold is 250 input connections and we have a thousand SV1 instances in Figure 1(d), for each one of the three SV2 instances, we would introduce four partial aggregate vertices, each working on one fourth of the SV1 instances, based on the network topology.

[1]To simplify the presentation, we do not discuss mechanisms that dynamically expand or contract the graph at runtime.

2.5 Structured Streams

In Scope, structured data can be efficiently stored as structured streams. Like tables in traditional databases, a structured stream has a well-defined schema that every record follows. Additionally, structured streams can be directly stored in a partitioned way, which can be either hash- or range-based over a set of columns. Data in a partition is typically processed together (i.e., a partition represents a computation unit). Each partition may contain a group of extents, which is the unit of storage in Scope. In the example of Figure 1(c), data is read from a structured stream *input.ss*, which is *not* partitioned by `nGram` (and therefore a repartition operator is needed). If the input structured stream were partitioned by `nGram` already, the resulting plan would consist of a single computation unit SV1, on which the final partitioning operator is replaced by the output operator. Relying on pre-partitioned data significantly reduces latency by removing both the data exchange and the superfluous global aggregate operators.

2.5.1 Indexes for Random Access

Within each partition, a local sorting order is maintained through a B^+-Tree index. This organization not only allows sorted access to the content of a partition, but also enables fast key lookup on a prefix of the sorting keys. Such support is very useful for queries that select only a small portion of the underlying data, and also for more sophisticated strategies such as index-based joins. In our example, if *input.ss* were partitioned and sorted by `nGram`, we could not only remove SV2 as explained earlier, but also the sort operator in SV1, with an additional improvement in performance.

2.5.2 Data Affinity

Scope does not require all the data that belongs to a partition to be stored in a single machine. Instead, Scope attempts to store all the data in a partition close together by utilizing *store affinity.* Every extent has an optional *affinity id.* All the extents with the same *affinity id* belong to an affinity group. The system tries to place all the extents of an affinity group on the same machine unless the machine has already been overloaded. In this case, the extents are placed in the same rack (or a close rack if the rack itself is overloaded). Each partition of a structured stream is assigned an *affinity id.* As extents are created within the partition, they get assigned the same *affinity id*, suggesting that they should be stored together. Processing a partition can be done efficiently either on a single machine or within a rack. Store affinity is a very powerful mechanism to achieve maximum data locality without sacrificing uniform data distribution.

The store affinity functionality can also be used to associate/affnitize the partitioning of an output stream with that of a *referenced* stream. This causes the output stream to mirror the partitioning choices (i.e., partitioning function and number of buckets) of the referenced stream. Additionally, each partition in the output stream uses the *affinity id* of the corresponding partition in the referenced stream. Therefore, two streams that are referenced not only are partitioned in the same way, but partitions are physically placed close to each other in the cluster. This layout significantly improves parallel join performance, as data need not be transferred across the network.

3. PARTIAL DATA REPARTITIONING

As described in Section 2.3, data partitioning typically consists of one or two physical operators: a partition operator, which splits its input into local partitions, and/or a merge operator, which collects and merges the local partitions that belong to the corresponding output partition. In absence of additional information, as illustrated in Figure 2, every merge vertex needs to connect and read data from every partition vertex. As we discuss in this section, however, by carefully defining the partition scheme of the exchange operator (e.g., number of partitions and partition boundaries), we can guarantee that certain local partitions would be empty. Additionally, we can somewhat influence how much data has to be transferred, and to which destination.

This general approach has several advantages during execution. First, by carefully defining partition boundaries, we can drastically reduce data transfer between partition and merge vertices by taking data locality into account. Second, partition vertices need to reserve fewer memory buffers and storage due to local partitions that are guaranteed to be empty. Third, the job manager does not need to maintain explicit connection state between partition and merge vertices for which the local partitions are empty, which reduces the footprint of the scheduler. Fourth, merge vertices do not have to wait for all partition vertices to finish executing, but only for those that actually contribute to the corresponding partition. This is important because a single partition outlier can delay all merge vertices from starting, increasing overall latency. Finally, fewer input connections to merge vertices reduce the chance of failures, thus requiring fewer intermediate aggregates (see Section 2.4.1), which improves overall performance.

In this section we explore efficient alternatives to perform data repartitioning that leverage properties of the *input* data to be repartitioned. Depending on the partitioning scheme, we discuss how to construct and implement data exchange operators that are defined over effective partitioning boundaries and only require a subset of connections between partition and merge vertices. Additionally, we discuss how to integrate these alternatives during query optimization. To illustrate the different partitioning techniques, we use the simple script below:

```
SELECT a, UDAgg(b) AS aggB
FROM SSTREAM "input.ss"
GROUP BY a;

OUTPUT TO SSTREAM "output.ss"
       [HASH | RANGE] CLUSTERED BY a;
```

The input in the script is a structured stream distributed in the cluster. The script performs a user defined aggregation UDAgg on column a and outputs the result into another structured stream, either hash- or range-partitioned by a.

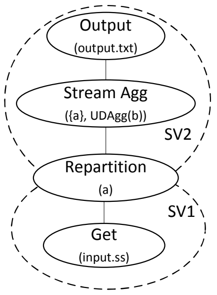


Figure 3: Execution Plan for a Simple Script

Figure 3 shows an execution plan for the script. Assuming that the input is not already partitioned by column a, the plan reads the input in parallel and hash-repartitions the data on column a ($SV1$ in the figure), and then merges the partitions, performs the user defined aggregate and outputs the result ($SV2$ in the figure). Note that in our example UDAgg is not associative nor commutative, so we cannot use local/global aggregates as in Figure 1(c)). If the input were already partitioned by a, a different execution plan without repartitioning would be preferable (i.e., read each partition, perform the aggregation, and write results in parallel).

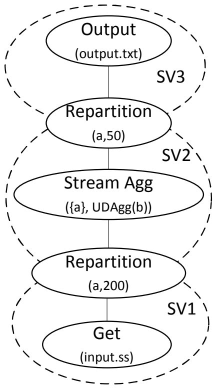


Figure 4: Partial Hash Repartitioning

Suppose that the input is indeed hash-partitioned by column a into 100 partitions. However, the user defined aggregate is very expensive, so we choose to repartition the input into 200 buckets. Additionally, the output of the aggregate is smaller than the input, so we choose to additionally repartition it into 50 buckets before the final output, in order to avoid writing too many small fragments. The resulting execution plan is shown in Figure 4. In principle, the execution plan looks rather expensive due to two full repartition operators. However, in this scenario there are important optimization opportunities. In fact, repartitioning a 100-way partitioned input into 200 partitions (by the same column) can be done by *locally* splitting each of the original partitions into two, without any network data transfer. Also, repartitioning a 200-way input into 50 partitions can be done by *partially* merging the input partitions. We next formalize this approach, and characterize when it is effective.

3.1 Hash-based Partitioning

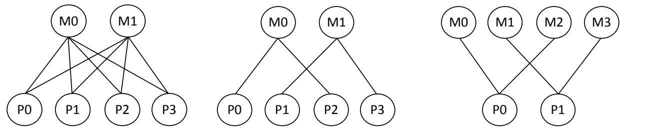
We consider the case in which the script writes a structured stream hash-partitioned by column a. We assume hash partitioning is done by first applying a hash function to the partitioning columns and then having the hash value modulo the number of output partitions to generate the partition number to which the row goes. As long as the hash function and the number of output partitions are fixed, hash partitioning is deterministic.

3.1.1 Determining Data Flow Connections

Suppose that we want to hash partition an input T into po partitions, and T is already hash-partitioned into pi partitions by the same columns. The naive execution plan contains pi vertices $P_0, \ldots P_{pi-1}$, each one partitioning its input into po local partitions, followed by po merge vertices

$M_0, \ldots, M_{po-1}$, each one reading and merging the *i-th* local partition from each of the pi partition vertices. Interestingly, for certain values of pi and po, some local partitions in P_i vertices would always be empty and do not need to be read by M_j vertices.

Example 1 *Suppose that $pi = 4$ and $po = 2$ (i.e., we want to partition 2-ways an input that is already 4-way partitioned). Every row in P_0 satisfies $h(C) \equiv 0 \mod 4$, where h is the hash function and C are the partitioning columns. Figure 5(a) shows the default partitioning strategy which connects every input vertex with every output vertex. In this case, we know that $h(C) \equiv 0 \mod 2$ as well, and therefore P_0 would never generate a row satisfying $h(C) \equiv 1 \mod 2$. Thus, M_1 does not need to read the* empty *local partition produced by P_0. In general, M_0 only reads from P_0 and P_2, and M_1 from P_1 and P_3. Figure 5(b) shows the refined merge graph. A similar strategy can be applied when $pi = 2$ and $po = 4$. Figure 5(c) shows the refined partitioning graph.*



(a) Full Partitioning (b) Partial Merge (c) Partial Partitioning

Figure 5: Different Hash Partitioning Strategies

In general, merge vertex M_i $(0 \leq i < po)$ needs to read from partition vertex P_j $(0 \leq j < pi)$ if there might be a row in P_j for which its hash value modulo po is i. In other words, if there exists an integer k such that $k \equiv j \mod pi$ and $k \equiv i \mod po$. By definition, this implies that there are integers k_1 and k_2 such that $k = k_1 \cdot po + i$ and $k = k_2 \cdot pi + j$, and therefore $k_1 \cdot po + i = k_2 \cdot pi + j$, or

$$po \cdot k_1 + (-pi) \cdot k_2 = (j - i)$$

This is a linear diophantine equation of the form $a \cdot x + b \cdot y = c$, which has integer (x, y) solutions if and only if c is a multiple of the greatest common denominator of a and b. In our case, there are solutions, and therefore M_i needs to reads from P_j if and only if $(j - i)$ is a multiple of $\gcd(po, -pi)$, or, more concisely, if and only if

$$i \equiv j \mod \gcd(pi, po)$$

If pi and po are co-primes, then $\gcd(pi, po) = 1$ and the optimized technique is the same as the traditional one (i.e., we need to connect every partition and merge vertex instances).

3.2 Range-based Partitioning

The ideas in the previous section can also be applied to range partitioning scenarios. Consider again the script in the previous section when it writes a structured stream range-partitioned by column a. Suppose now that the input in the figure is range-partitioned by columns (a, b). Note that this property does not imply the data being partitioned by a alone, since two rows that share a values might be in different partitions due to varying b values. In this case, the same plan of Figure 3 can certainly be used, with a full repartitioning on a. However, an input that is range-partitioned by (a, b) can be repartitioned by a in a cheaper way by *locally* splitting and merging the original partitions, as shown next.

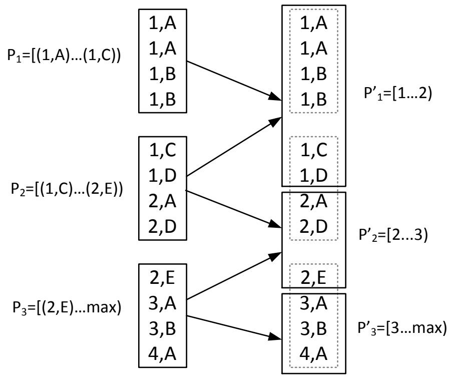


Figure 6: Refining Range Partitions from (a, b) to a

Example 2 *Consider the data set at the left of Figure 6, which is range-partitioned by columns (a, b) into three partitions. Partition P_2, for instance, contains all rows with values in the interval $[(1, B), (2, D))$ (note that the interval is closed on left and open on right). As shown in the figure, we can repartition this data set by column a without connecting each partition output with each merge input. The precise connectivity map can be statically computed at compile time given the original partitioning boundaries. Note that the output partition P'_1 receives data from input partitions P_1 and P_2. While P_1 is fully contained into P'_1, we need to filter rows in P_2 with $a < 2$ to avoid incorrect results. Compared to the initial* physical *partitions P_i, we call P'_i as* logical *partitions, each of which conceptually reads one or more physical partitions with optional filtering predicates. The end result is that P'_i is range-partitioned on column a.*

In general, partial range-based repartitioning can be applied whenever the input and output partition schemes share a common column prefix (otherwise, there is no choice but to connect each partition vertex with every merge vertex). The general approach consists of two steps. First, we need to determine the range boundaries for each output partition. With this information, we can then generate code that determines the output partition for each incoming row, and determine which partition and merge vertices are connected. We next discuss these steps in detail.

3.2.1 Determining Partitioning Boundaries

The main goal of query optimization is to minimize the resulting job latency. In the context of range partitioning, and especially with respect of boundary determination, there are two main aspects that contribute to overall latency. First, the resulting partitions need to be evenly distributed. Any skewed partition is likely to introduce outliers during execution, as a partition that is significantly larger than the average increases the latency of the overall job (and likely propagates skewed partitions to subsequent execution vertices). Second, the cost of the repartitioning operator itself is important, which is directly proportional to the amount of network data transfer. As a consequence, we should try to align input and output partition boundaries as much as possible. This way, we avoid splitting input partitions in multiple large fragments that are sent across the network.

Suppose that we want to repartition data on columns C, where each partition is roughly of size T. When the input is already partitioned by columns C' (where C and C' share a common prefix), the input partitioning metadata information itself can be viewed as a (coarse) histogram B over columns C', with each partition representing a bucket in the histogram[2]. We assume that buckets maintain the number of rows that are included in its boundary, that bucket boundaries are closed on left and open on right, and that the last range on right is a special *maximum* value that is larger than any other valid value. The information from such histograms can help the system choose partition boundaries.

Some very common column types (e.g., strings or binary columns) are not amenable to interpolation, which limits what we can infer about value distributions inside a histogram bucket. Specifically, in this case we know all bucket boundaries but we cannot interpolate and generate intermediate values within input buckets. Under this natural restriction, Algorithm 1 describes a greedy approach to determine partition boundaries that are compatible with C and would result in partitions of size around T.

Algorithm 1: PartitionBoundaries(C, T, B)

```
Input:  Columns C, Partition size T, Buckets B
Output:  Partition boundaries P
/* Assume that C and B.cols share common prefix CP
   and for each $1 < i <$|B|:  B[i-1].hi = B[i].lo    */
/* Output is partitioned by CP, which implies C,
   and each partition size is around T              */
CB = $\cup_i$[$\Pi_{CP}$B[i].lo, $\Pi_{CP}$B[i].hi) // project B on CP
idx = 0;
while idx < |CB| do
    actLo = CB[idx].Lo;
    actSize = CB[idx].Size;
    idx++;
    while actLo = CB[idx].Lo OR
          CB[idx].size / 2 < T - actSize do
        actSize += CB[idx].size;
        idx++;
    end
    P = P $\cup$ [actLo, CB[idx-1].hi);
end
return P;
```

First, we project the histogram buckets onto the common prefix between input and output partitioning columns. Then, we walk the projected buckets and accumulate such buckets into new output partitions. We keep accumulating buckets into a new partition as long as (i) the lower bound of the bucket is the same as the current lower bound of the partition (which can happen when we project the original bucket boundaries onto the common prefix), and (ii) the resulting bucket is as close as possible to the target size T. Note that due to bucket granularity and the need to collapse together buckets that share the same lower bounds, the resulting partitions might not exactly match the expected size T. This data skew is known at compile time and taken into account during optimization, as we discuss later.

In some cases, however, there is additional information that we can leverage. For instance, if the input data comes from a structured stream, the optional local B^+-tree-like index provides detailed distribution of values within each input partitions. Also, there could be a histogram obtained from statistics on the relevant columns. Finally, for column types that are amenable to interpolation (e.g., integer or float), a uniform data distribution within each partition can be assumed. In these cases, we can additionally leverage such value distribution and further improve the quality of the resulting partitions by slightly extending Algorithm 1.

[2]Input partitioning metadata is obtained directly for base structured streams, and propagated throughout query plans analogously to other properties such as cardinality information.

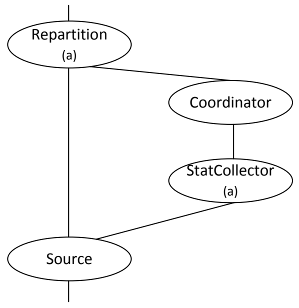


Figure 7: Getting Partition Boundaries at Runtime

Partition boundaries can be chosen not only at compile time but also at runtime. If the optimizer detects potentially unbalanced partition boundaries, it also considers an alternative plan that uses a more expensive repartitioning operator at runtime but results in good-quality partition boundaries. Figure 7 shows how to achieve this goal, by using additional computation stages. We first intercept the input rowset and compute a histogram on the partitioning columns (*StatCollector* operator in the figure). This intermediate information is aggregated by a *Coordinator*, which obtains the global histogram for the partitioning columns. The coordinator then computes the global partition boundaries as before, and broadcast them to the actual partition vertices, which additionally receive the input to be partitioned, and are executed in the traditional way. The optimizer considers this alternative along with the ones defined in the previous section and chooses the one that is the cheapest in estimated costs, considering both the local partitioning cost and the increase in latency by unbalanced partitions.

3.2.2 Determining Data Flow Connections

Once the partition boundaries are calculated, it is straightforward to determine which output partition and input merge vertices should be connected to each other. Suppose that $[PO^i_{lo}, PO^i_{hi})$ is the *i-th* partition range boundary (i.e., it corresponds to the *i-th* merge vertex in the execution graph). We should connect this merge vertex with a partition vertex defined over the input range $[PI^j_{lo}, PI^j_{hi})$ whenever:

$$\Pi_{CP}[PO^i_{lo}, PO^i_{hi}) \cap \Pi_{CP}[PI^j_{lo}, PI^j_{hi}) \neq \emptyset$$

where CP is the longest common prefix between the output and input partitioning columns, and the projection of a range with columns C is defined as follows:

$$\Pi_{CP}[lo, hi) = \begin{cases} [lo, hi) & ifCP = C \\ [\Pi_{CP}(lo), \Pi_{CP}(hi)] & otherwise \end{cases}$$

Additionally, we add a filter predicate that keeps incoming rows in the appropriate range unless we can guarantee that all input rows qualify, which happens whenever:

$$\Pi_{CP}[PI^j_{lo}, PI^j_{hi}) \subseteq [PO^i_{lo}, PO^i_{hi})$$

The range predicate can be implemented by exploiting an index if available, or otherwise by scanning an input partition and removing extra rows on the fly. This choice is known at compile time and incorporated into the cost model.

3.3 Optimizer Integration

As discussed in Section 2, structured streams are partitioned when stored in the cluster. The structured stream partitioning scheme might or might not be compatible with that of subsequent repartition operators. Defining the partitioning scheme as well as which columns to partition on is part of the physical data design problem, which is crucial in traditional database systems and is also gaining importance in distributed computation engines. Partitioning choices are generally made based on usage patterns, as there is no optimal physical design independent of a workload.

In this section, we discuss how to integrate different partition strategies into the optimization framework, particularly in the presence of structured streams, and identify some optimization opportunities. The optimizer considers all the alternatives in a uniform framework and chooses the optimal solution based on the estimated costs. The optimizer search space increases by adding new partitioning strategies. The exact overhead heavily depends on queries and physical design but is reduced by effective cost-based pruning. Overall, we only observed marginal increase in compilation time in the cluster due to these extensions. We also note that our environment has relaxed requirements on optimization time compared to traditional database systems, so a slight increase in optimization time is acceptable.

We enhance the traditional optimizer cost model [24] in two main directions. First, in addition to CPU and I/O cost, each operator estimates network transfer based on the amount of data to be shuffled. Second, we adjust the impact of partition skew by explicitly charging each operator the cost of the largest partition it needs to process. We omit details due to space constraints.

3.3.1 General Repartitioning Opportunities

The Scope optimizer chooses to repartition data based on requirements from subsequent operators. For instance, a group-by operator would cause the optimizer to require its input to be partitioned by a subset of the grouping columns. To determine the number of partitions, the optimizer considers the total data size that needs to be processed, and also the cost to process each row. The goal of this estimation is to obtain partitions that are processed in a reasonable amount of time (partitions should not be too large, which increases recovery time, nor too small, which increases scheduling and vertex start-up overhead).

If the input is partitioned by a different but compatible set of columns, the optimizer considers partial partitioning strategies besides the full repartitioning alternative. Additionally, if the input is partitioned by the right columns but with a different number of partitions (or different ranges for range partitioning), the optimizer considers alternatives that, while not optimal with respect to partition size, might result in a faster repartitioning and cheaper overall plan cost. For instance, suppose that the optimizer determines that the number of partitions should be 100. However, the input is already hash-partitioned by the right columns into 45 partitions. The optimizer would also consider an alternative with 90 partitions, which would result in slightly larger partitions but a more efficient repartition operation.

3.3.2 Opportunities for N-ary Operators

Operators that receive multiple inputs, such as union and join variants, have an important requirement: all their inputs need to be partitioned in the same way to prevent wrong results. For instance, suppose we are joining two inputs R and S on columns $(R.a, R.b, R.c) = (S.d, S.e, S.f)$. The result would be correct whenever we partition both inputs on any subset of these columns, as long as we use the same subset on each input (modulo join equality). Otherwise, rows that would join might end up in different partitions and we would miss rows in the result. We next discuss some optimization opportunities for n-ary operators:

Pushing partition schemes from one input to others. If a join input is already partitioned by a subset of the join columns, we attempt to use such partitioning scheme for the other input. That way, if the other input is partitioned in a compatible way (e.g., same columns but different number of partitions for hash partitioning, or common column prefix for range partitioning) we might be able to use a cheaper partial repartition operator on the remaining inputs, leading to a better plan overall.

Heuristic range partitioning. Suppose that all inputs of an n-ary operator are range-partitioned by the same columns, but with different ranges. If these inputs are defined over different domains of data, there might not be a single partition among inputs to *push* to the others without resulting in large partition skew. To address this scenario, the optimizer additionally obtains a common partitioning scheme that takes into account the overall distribution of inputs. We first collect all histogram buckets from each input, union them together, and use a slightly refined version of Algorithm 1 to obtain the overall required partition boundaries. Finally, we push this partitioning requirement to each input.

Broadcast optimization. When one side of the join is very large and the other is small, it is common to rely on a broadcast join variant, where the small input is sent to all partitions of the large input. As the size of the small input increases, this strategy loses its competitive advantage. Interestingly enough, if both inputs are partitioned in such a way that there is a common prefix between the partitioning and join columns, we can improve the traditional broadcast join optimization and handle larger "small inputs" as follows. We project the partition boundaries from the big input on the common prefix as the required partition boundaries for the small input. Rather than sending the whole small input to each partition of the large input, we determine, for each partition p in the large input, all the partitions in the small input that have rows which might join with rows in p. In this situation, we might send data from the small input multiple times to different partitions from the big input but avoid the expensive full repartitioning altogether.

3.3.3 Eliminating Repartitioning

An extreme case of repartitioning optimization happens when the optimizer is able to remove a partitioning operator completely leveraging additional data properties. Suppose that we want to partition the input data on column a into 100 partitions, and the input is already hash-partitioned by column b on the same number of partitions. If there is a functional dependency $b \to a$, then we know that the input is also partitioned by column a and we completely eliminate a repartition operation. As another example, if the input is partitioned by (a, b) but we can determine that b is a constant (e.g., perhaps due to some filter operation), the input is effectively partitioned by a.

It is crucial to note, however, that while these schemes correctly partition data by column a, they are not the same partitions that we would obtain by directly hash partitioning by a alone. To illustrate this point, suppose that $a = b + 1$, and so $b \rightarrow a$. Every row belongs to partition $h(b) \mod 100$, which is not the same as $h(a) \mod 100 = h(b+1) \mod 100$. In some scenarios we can leverage this partitioning scheme (e.g., if the partitions are consumed by a group-by operator), but not in other scenarios (e.g., if the partitions are consumed by an n-ary operator that requires the same partitioning scheme on all its inputs). The optimizer is aware of the context of each partitioning request and can determine when fully eliminating a partitioning operation based on functional dependencies or constraints is possible.

3.3.4 Other Considerations

Skewed partitions in input structured streams or intermediate results during query execution can have a very negative impact on query performance. A partition that is significantly larger than average increases the overall job latency. On the other hand, too many small partitions may increase the number of vertices and overall scheduling overhead. Similar to Algorithm 1, we consider refining partition boundaries both at compilation time and at runtime. In this case, the refined partitions are still partitioned by the same columns but with more even distribution.

Finally, Algorithm 1 assumes the input and the output partition columns share a common prefix. The same concept applies to string prefix optimization. That is, if the input data is range-partitioned by a string column s, we can partially partition the data by a *prefix* of column s.

4. INDEXED-BASED PARTITIONING

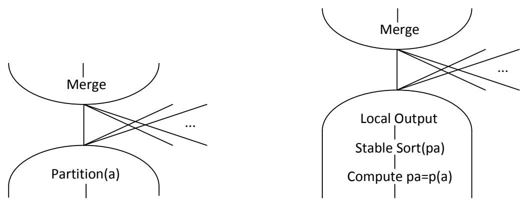
Consider again the example of Figure 1, but suppose that there are several terabytes of input data (and therefore several thousand SV1 vertex instances). In that case, we need to fully repartition the input into, say, ten thousand partitions (i.e., ten thousand SV2 vertices). Each SV1 vertex needs to partition its output into ten thousand different streams, which are later read and merged by SV2 vertices. When the number of partitions is very large, this procedure presents some scalability challenges. In this section we introduce an alternative way to repartition data that leverages structured stream technology (see Section 2.5 for details), has similarities with the duality between sorting and hashing that was discussed in [10], and can be applied to both range- and hash-based partitioning schemes.

In the example in Figure 1, rows arrive to the repartition operator sorted by `nGram`. However, the hash function effectively randomizes the partition that each row belongs to. To avoid constant random writes to local storage for each partitioned value, we need to keep one buffer in memory for each partition and flush the buffer when it gets full. For a large number of partitions, the combined memory requirements for such buffers start interfering with those of other operators. More importantly, the underlying store is optimized for large volumes of data and sequential access. This goal is achieved by reserving a minimum amount of sequential space in disk for each stream (in the order of megabytes) and making it the append and compression unit. This means that we would have to reserve a very large amount of space in the local disk volume to support all the local partitions, which not only decreases performance, but wastes significant amount of storage for the duration of the job.

An alternative approach, which we call *index-based partitioning*, can be used to significantly improve I/O efficiency. Suppose that we want to repartition a rowset by column a (Figure 8(a) shows the traditional way of implementing this operation). For each partition vertex, instead of writing to a different stream per partition, we change the operation as follows (see Figure 8(b)). First we introduce a new computed column, called pa, to the input rowset that needs to be partitioned. The value of such column is equal to $p(a)$, the *partition number* that would be assigned to the corresponding row with value a (e.g., for hash-based partitioning into N partitions with hash function h, $p(a) = h(a) \mod N$, and for range-based partitioning $p(a)$ is the index of the bucket that contains value a).

Then, we insert a *stable sort* operator on column pa. A stable sort operator sorts the input rowset by pa and keeps the original row order in case of ties. Therefore, the output of such sort contains all input rows grouped together by the partition they belong to, and within each partition the rows satisfy an ordering consistent to the original one. This is very important as the optimizer can continue to exploit the original input ordering for the following operations. In other words, this approach produces the same result as if we actually created all partitions in the default way and concatenate them together in order of the partition number.

Finally, we store this "partitioned" result locally as a structured stream, with a B$^+$-tree index on column pa. The merge operation stays the same, but reading data from this structured stream with a specific key value. Instead of locating the corresponding file and streaming all the rows, we perform a key lookup of the partition number on the B$^+$-tree and return all relevant rows.



(a) Traditional partitioning. (b) Indexed partitioning.

Figure 8: Scalable Indexed-based Partitioning

The indexed-based approach as discussed above operates outside of the optimizer. That is, after the optimizer determines that repartitioning is required, we can decide in a post-processing step whether to implement the operator in the traditional or index-based manner. This choice is necessary because each alternative might be the faster one. On one hand, index-based partitioning improves I/O efficiency for partition operators. At the same time, it performs B$^+$-tree lookups to return the corresponding partitions, and more importantly, it introduces an additional stable sort operation. This is especially wasteful when the stable sort is placed directly or indirectly on top of another sort operation, due to the following equivalence:

$$\text{Stable-Sort}_b(\text{Sort}_a(R)) \equiv \text{Sort}_{b,a}(R)$$

We incorporate index-based partitioning into the optimizer in SCOPE in a principled way. Particularly, we enhance the set of enforcer rules [9] in the SCOPE optimizer. Every time we *enforce* a partitioning request, we generate both a traditional repartitioning operator and an indexed-based alternative. The indexed-based alternative consists of a compute scalar operator for a new column p as in Figure 8(b) directly below the index-based partitioning operator (note that we do not explicitly insert a stable sort on p as in the figure). Instead, the index-based partitioning operator requires its input to be sorted by (p, X) where X is the sort requirement (if any) of the original partitioning operator. We then let the optimizer explore different alternatives, by relying on transformation rules that, for instance, exploit the sort equivalence described above, take advantage of functional dependencies (since p is functionally determined by the partitioning columns) and reorder operators (e.g., pushing the compute scalar operator and corresponding sorts down the tree). This approach lets the optimizer choose, in a cost-based manner, the most efficient execution plan including both alternatives in Figure 8, but other options as well.

5. EXPERIMENTAL EVALUATION

We implemented all the advanced partitioning techniques and their optimization in Scope, which is deployed on production clusters consisting of tens of thousands of machines at Microsoft. Tens of thousands of jobs are executed daily, reading and writing tens of petabytes of data in total, and powering different online services.

In this section, we perform experimental evaluation of different techniques and report results in a small test cluster of a hundred machines. Each machine has two six-core AMD Opteron processors running at 1.8GHz, 24 GB of DRAM, and four 1TB SATA disks. All machines run Windows Server 2008 R2 Enterprise X64 Edition. The results correlate very well with observations we made on production clusters consisting of thousands of machines. Due to confidential data/business information, we report performance trends rather than actual numbers for all the experiments.

5.1 Partial Partitioning

In the context of web data analytics, it is common to calculate aggregates of raw data at different granularities (e.g., averages per domain, per host, and so on). Consider a large structured stream that contains information about web pages. A typical schema for this data set is shown in Table 1, where each page URL is decomposed into the (reversed) domain, (reversed) host, top-level-directory, and URL-suffix columns. The data is range-partitioned and sorted by *(domain, host, top-level-directory)* in a structured stream, so that related pages are clustered together. Due to very skewed domain distributions, we cannot partition the data by *domain* alone, or we would get partitions with very large number of rows (the same trend is observed when partitioning by $(domain, host)$).

Consider now a query that groups the input table by domain and host values, and performs several user-defined aggregates over the remaining columns [3]:

```
SELECT domain, host, Agg(col1), ..., Agg(coln)
FROM SSTREAM "WebPages.ss"
GROUP BY domain, host
```

[3]This is a simplified version of a family of pipelines that are very common in the context of web-experiments. A large fraction of our production cluster utilization goes into executing pipelines that use this pattern in one way or another.

Since the input is partitioned by a superset of the grouping columns, we need to repartition the input to the group-by operator. Figure 9 shows the results of evaluating this query under two alternative execution plans: one which performs a traditional full repartition on columns $(domain, host)$, and another that applies the partial repartitioning approach discussed in Section 3.2. We can see in Figures 9(a) that using partial repartitioning significantly improves query latency by a 3.5x speedup. We also calculate the total work for each query by adding up latencies for each vertex. As shown in Figure 9(b), partial repartitioning reduces the total work by over 7 times. The improvement in latency is less than that on total work, because the strategy that uses partial repartitioning sometimes requires reading slightly larger partitions due to boundary alignment and overlap (see Algorithm 1).

To understand the performance gain, Figure 9(c) shows the fraction of I/O incurred by the new alternative, compared with the traditional full repartitioning, in both data write and data read, respectively. The traditional approach has to read and write the *whole* data set when performing a full repartitioning, and then read the repartitioned data again. In contrast, the new approach avoids an intermediate write/read operation and directly reads the right partitions, using an appropriate filter that is evaluated efficiently by exploiting the underlying index. Besides the savings in I/O, the new approach has significant fewer vertices (e.g. no explicit partition and merge vertices) and does not have the heavily connected scheduling graph between the partition and merge vertices as in the full repartitioning case, both of which greatly improve scheduling efficiency at runtime.

5.2 Optimizing N-ary Operators

We next show a more complex example of partial range partitioning in the case of a N-ary operation, where obtaining the right partition boundaries is crucial for performance. This scenario is similar to the previous one, with an important difference. Rather than having a single input stream, we have four sources of data, each one obtained in a different period of time over a different domain. Although each data source is range-partitioned by the same columns, the corresponding partition boundaries are different due to input sources being inherently biased in different ways. The modified query, which unions together all input sources before aggregating the result, is shown below:

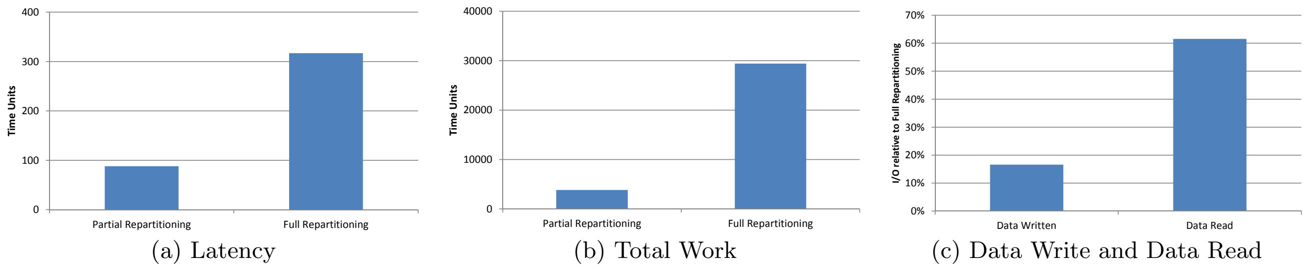
```
SELECT domain, host, Agg(col1), ..., Agg(coln)
FROM (
    SELECT * FROM T1 UNION ALL
    SELECT * FROM T2 UNION ALL
    SELECT * FROM T3 UNION ALL
    SELECT * FROM T4
     )
GROUP BY domain, host
```

Figure 10 shows three different execution alternatives: *partial repartitioning*, or *PR* (which uses the technique described in Section 3.3.2 to obtain a global set of bucket boundaries), *full partitioning*, or *FR* (which performs the traditional partitioning), and *partial repartitioning without boundary merge*, or *PRN* (which, instead of considering *all* inputs for choosing partitioning boundaries, uses the partitioning boundaries of *one* input to partially repartition the others).

Figures 10(a-b) show that *PR* improves both the latency and total work of the query by a 6x factor compared to *FR*. Specifically, Figure 10(c) shows that *PR* incurs in around

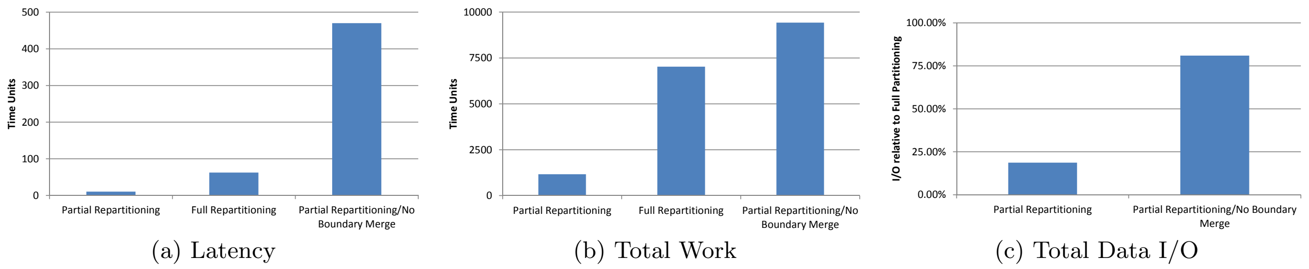
Domain	Host	Top-level-directory	URL-suffix	Data
com.microsoft	www	download/	en/default.aspx?WT.mc_id=MSCOM_HP_US_Nav_Downloads	...
com.microsoft	windows	products/	home	...
com.bing	www	videos/	browse?FORM=Z9LH6	...
...	...	...	...	...

Table 1: Sample Information for a Web-pages Structured Stream



(a) Latency (b) Total Work (c) Data Write and Data Read

Figure 9: An Aggregation Query over Web-pages



(a) Latency (b) Total Work (c) Total Data I/O

Figure 10: A Union-All Query on Web-pages

20% total I/O compared to *FR*. The ratio is smaller than 50% because the *FR* alternative needs to perform intermediate aggregate stages as described in Section 2.4.1.

It is also interesting to discuss the relative performance of the *PRN* strategy. Although the total I/O for *PRN* is less than 80% compared to that of *FR*, the total work and latency are much worse for *PRN*. In particular, the latency of *PRN* is 7 times longer than that of *FR*. This non-intuitive behavior is explained when looking more carefully at the input data. When reusing the partition boundaries of one input to repartition and union all inputs together, we introduce a big source of data skew. Some partitions are much smaller and others much larger than the average. While this observation does not significantly affect the total amount of work done, it substantially increases latency. It illustrates the importance of optimizing partitioning boundaries in order to improve the overall query performance.

5.3 Indexed-based Partitioning

To measure the benefits of indexed-based partitioning, we executed a script that requires repartitioning on a single column, and vary the number of output partitions from 250 to 4,000. Figure 11 shows the average latency of a partitioning vertex, normalized to the time it takes the traditional technique to do a 250-way repartitioning. We can see in the figure that traditional partitioning performance degrades visibly when increasing the number of partitions, due to increased random I/Os resulting in poor I/O performance. Not shown in the figure, but also important, is the fact that the amount of wasted storage due to reserved and unused disk also increases with the number of partitions. On the other hand, we can see that indexed-based partitioning has a longer, fixed start-up cost due to the additional sort by partition id, and it is worse than the traditional approach for fewer than 500 partitions. However, the performance of the index-based approach is almost independent on the number of partitions (it slightly increases when adding more partitions due to a larger index structure in the structured stream). Indexed-based partitioning achieves parity with the traditional approach at around 500 partitions and is already 1.6x faster when using 4,000 partitions.

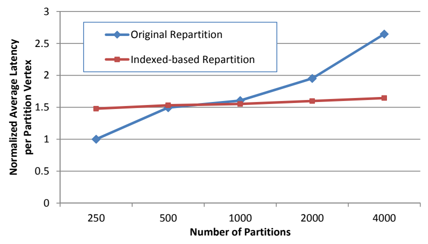


Figure 11: Partitioning Scalability

6. RELATED WORK

The last decade witnessed the emergence of various solutions for massive distributed data storage and computation. Distributed file systems, such as Google File System [7], Hadoop Distributed File System [1], and Microsoft Cosmos [3], provide scalable and fault-tolerant storage solutions. Google's MapReduce [5] provides a simple abstraction of common group-by-aggregation operations where *map* functions correspond to groupings and *reduce* functions correspond to aggregations. Microsoft's Dryad [12] provides additional flexibility where a distributed computation job is represented as a dataflow graph, and supports scripting languages that allow to easily compose distributed data-flow programs. These programming models help developers write distributed applications, but require dealing with implementation details to achieve good performance.

To tackle these shortcomings, high level programming languages plus conceptual data models were recently proposed, including Jaql [2], Scope [3, 24], Tenzing [4], Dremel [15], Pig [18], Hive [21], and DryadLINQ [23]. Regardless of the language differences, their declarative nature hides system complexities from the users and can benefit from an optimizer to generate efficient execution plans.

Scope relies on a cost-based query optimizer [24] that considers performance tradeoffs of the entire system, including language, runtime, and distributed store. It leverages database optimization techniques and also incorporates unique requirements derived from the context of distributed query processing. Scope also supports data in both unstructured and structured formats. Rich structural properties and access methods from structured streams provide many unique opportunities for efficient physical data design and distributed query processing. Previous work [24] mentioned "refined partitioning" and "refined merge" but didn't offer the details. To the best of our knowledge, this is the first paper to discuss advanced partitioning strategies, based on input data properties and index strategies, and fully integrate the reasoning into a uniform optimization framework.

Many of the core optimization techniques that the Scope optimizer implemented originated from early research in parallel databases [6, 13] and traditional databases [11, 14, 16, 17, 19, 20, 22]. The Scope optimizer enhances previous work in several ways, such as by fully integrating parallel plan optimization and reasoning with more complex properties derivation relying on structural data properties. Our work in this paper enables the optimizer to consider additional partitioning strategies and come up with more efficient query execution plans.

7. CONCLUSION

Massive data analysis in cloud-scale data centers plays a crucial role in making critical business decisions and improving quality of service. 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. Data shuffling is the most expensive operation in such environment. Its judicious placement and implementation techniques play a vital role in the effectiveness and efficiency of cloud-scale query execution. We describe several advanced partitioning techniques to significantly improve data shuffling efficiency and integrate such complex reasoning into the query optimizer to generate much more efficient query plans. The system intelligently exploits the input data properties and performs partial partitioning by moving a only small subset of the input data set whenever possible. A novel index-based partitioning strategy is also used for the system to efficiently support a massive data partitioning operation that generates thousands of partitions. The techniques are incorporated in Scope, running over data clusters of tens of thousands of machines, and have proven to be effective, greatly improving query performance for a wide range of real-world jobs.

8. REFERENCES

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

[2] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan, and E. J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. In *Proceedings of VLDB Conference*, 2011.

[3] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. In *Proceedings of VLDB Conference*, 2008.

[4] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing: A SQL implementation on the mapreduce framework. In *Proceedings of VLDB Conference*, 2011.

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

[6] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database processing. *Communications of the ACM*, 36(6), 1992.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of SOSP Conference*, 2003.

[8] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proceeding of SIGMOD Conference*, 1990.

[9] G. Graefe. The Cascades framework for query optimization. *Data Engineering Bulletin*, 18(3), 1995.

[10] G. Graefe, A. Linville, and L. Shapiro. Sort versus hash revisited. *IEEE Transactions on Knowledge and Data Engineering*, 6(6), 1994.

[11] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proceeding of ICDE Conference*, 1993.

[12] 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*, 2007.

[13] A. Jhingran, T. Malkemus, and S. Padmanabhan. Query optimization in DB2 parallel edition. *Data Engineering Bulletin*, 20(2), 1997.

[14] H. Lu. *Query Processing in Parallel Relational Database Systems*. IEEE Computer Society Press, 1994.

[15] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of webscale datasets. In *Proceedings of VLDB Conference*, 2010.

[16] T. Neumann and G. Moerkotte. A combined framework for grouping and order optimization. In *Proceedings of VLDB Conference*, 2004.

[17] T. Neumann and G. Moerkotte. An efficient framework for order optimization. In *Proceedings of ICDE*, 2004.

[18] 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*, 2008.

[19] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of SIGMOD Conference*, 1979.

[20] D. Simmen, E. Shekita, and T. Malkenus. Fundamental techniques for order optimization. In *Proceedings of SIGMOD Conference*, 1996.

[21] 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*, 2010.

[22] X. Wang and M. Cherniack. Avoiding sorting and grouping in processing queries. In *Proc. of VLDB Conference*, 2003.

[23] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. of OSDI Conference*, 2008.

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