The VLDB Journal
DOI 10.1007/s00778-012-0280-z

SPECIAL ISSUE PAPER

SCOPE: parallel databases meet MapReduce

Jingren Zhou · Nicolas Bruno · Ming-Chuan Wu · Per-Ake Larson · Ronnie Chaiken · Darren Shakib

Received: 15 August 2011 / Revised: 16 February 2012 / Accepted: 14 May 2012

Abstract Companies providing cloud-scale data services have increasing needs to store and analyze massive data sets, such as search logs, click streams, and web graph data. For cost and performance reasons, processing is typically done on large clusters of tens of thousands of commodity machines. Such massive data analysis on large clusters presents new opportunities and challenges for developing a highly scalable and efficient distributed computation system that is easy to program and supports complex system optimization to maximize performance and reliability. In this paper, we describe a distributed computation system, Structured Computations Optimized for Parallel Execution (SCOPE), targeted for this type of massive data analysis. SCOPE combines benefits from both traditional parallel databases and MapReduce execution engines to allow easy programmability and deliver massive scalability and high performance through advanced optimization. Similar to parallel databases, the system has a SQL-like declarative scripting language with no explicit parallelism, while being amenable to efficient parallel execution on large clusters. An optimizer is responsible for converting scripts into efficient execution plans for the distributed computation engine. A physical execution plan consists of a directed acyclic graph of vertices. Execution of the plan is orchestrated by a job manager that schedules execution on available machines and provides fault tolerance and recovery, much like MapReduce systems. SCOPE is being used daily for a variety of data analysis and data mining applications over tens of thousands of machines at Microsoft, powering Bing, and other online services.

Keywords SCOPE · Parallel databases · MapReduce · Distributed computation · Query optimization

J. Zhou (✉) · N. Bruno · M.-C. Wu · P.-A. Larson · R. Chaiken · D. Shakib
Microsoft Corp., One Microsoft Way, Redmond, WA 98052, USA
e-mail: jrzhou@microsoft.com

N. Bruno
e-mail: nicolasb@microsoft.com

M.-C. Wu
e-mail: mingchuw@microsoft.com

P.-A. Larson
e-mail: palarson@microsoft.com

R. Chaiken
e-mail: rchaiken@microsoft.com

D. Shakib
e-mail: darrens@microsoft.com

1 Introduction

The last decade witnessed an explosion in the volumes of data being stored and processed. More and more companies rely on the results of such massive data analysis for their business decisions. Web companies, in particular, have increasing needs to store and analyze the ever growing data, such as search logs, crawled web content, and click streams, usually in the range of petabytes, collected from a variety of web services. Such analysis is becoming crucial for businesses in a variety of ways, such as to improve service quality and support novel features, to detect changes in patterns over time, and to detect fraudulent activities.

One way to deal with such massive amounts of data is to rely on a parallel database system. This approach has been extensively studied for decades, incorporates well-known techniques developed and refined over time, and mature system implementations are offered by several vendors. Parallel database systems feature data modeling using well-defined schemas, declarative query languages with high levels of abstraction, sophisticated query optimizers, and a rich runtime environment that supports efficient execution strategies.

Published online: 28 June 2012

At the same time, database systems typically run only on expensive high-end servers. When the data volumes to be stored and processed reaches a point where clusters of hundreds or thousands of machines are required, parallel database solutions become prohibitively expensive. Worse still, at such scale, many of the underlying assumptions of parallel database systems (e.g., fault tolerance) begin to break down, and the classical solutions are no longer viable without substantial extensions. Additionally, web data sets are usually non-relational or less structured and processing such semi-structured data sets at scale poses another challenge for database solutions.

To be able to perform the kind of data analysis described above in a cost-effective manner, several companies have developed distributed data storage and processing systems on large clusters of low-cost commodity machines. Examples of such initiatives include Google's MapReduce [11], Hadoop [3] from the open-source community, and Cosmos [7] and Dryad [31] at Microsoft. These systems are designed to run on clusters of hundreds to tens of thousands of commodity machines connected via a high-bandwidth network and expose a programming model that abstracts distributed group-by-aggregation operations.

In the MapReduce approach, programmers provide map functions that perform grouping and reduce functions that perform aggregation. These functions are written in procedural languages like C++ and are therefore very flexible. The underlying runtime system achieves parallelism by partitioning the data and processing different partitions concurrently.

This model scales very well to massive data sets and has sophisticated mechanisms to achieve load-balancing, outlier detection, and recovery to failures, among others. However, it also has several limitations. Users are forced to translate their business 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 difficult to reuse. Moreover, for applications that require multiple stages of MapReduce, there are often many valid evaluation strategies and execution orders. Having users implement (potentially multiple) map and reduce functions is equivalent to asking users 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. Hand-crafted execution plans are more often than not suboptimal and may lead to performance degradation by orders of magnitude if the underlying data or configurations change. Moreover, attempts to optimize long MapReduce jobs are very difficult, since it is virtually impossible to do complex reasoning over sequences of opaque MapReduce operations.

Recent work has systematically compared parallel databases and MapReduce systems, and identified their strengths and weaknesses [27]. There has been a flurry of work to address various limitations. High-level declarative languages, such as Pig [23], Hive [28,29], and Jaql [5], were developed to allow developers to program at a higher level of abstraction. Other runtime platforms, including Nephele/PACTs [4] and Hyracks [6], have been developed to improve the MapReduce execution model.

In this paper, we describe SCOPE (Structured Computations Optimized for Parallel Execution) our solution that incorporates the best characteristics of both parallel databases and MapReduce systems. SCOPE [7,32] is the computation platform for Microsoft online services targeted for large-scale data analysis. It executes tens of thousands of jobs daily and is well on the way to becoming an exabyte computation platform.

In contrast to existing systems, SCOPE systematically leverages technology from both parallel databases and MapReduce systems throughout the software stack. The SCOPE language is declarative and intentionally reminiscent of SQL, similar to Hive [28,29]. The select statement is retained along with joins variants, aggregation, and set operators. Users familiar with SQL thus require little or no training to use SCOPE. Like SQL, data are internally modeled as sets of rows composed of typed columns and every row set has a well-defined schema. This approach makes it possible to store tables with schemas defined at design time and to create and leverage indexes during execution. 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 data source, regardless of whether it is structured or not), *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 process both relational and non-relational data sets and solves problems that cannot be easily expressed in traditional SQL, while at the same time retaining the ability to perform sophisticated optimization of user scripts.

SCOPE includes a cost-based optimizer based on the Cascades framework [15] that generates efficient execution plans for given input scripts. 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 as a whole. The SCOPE optimizer extends the original Cascades framework by incorporating unique requirements derived from the context of distributed query processing. In particular, parallel plan optimization is fully integrated into the optimizer, instead of being done at the

post-optimization phase. The property framework is also extended to reason about more complex structural data properties.

The SCOPE runtime provides implementations of many standard physical operators, saving users from having implementing similar functionality repeatedly. Moreover, different implementation flavors of a given physical operator provide the optimizer a rich search space to find an efficient execution plan. At a high level, a script is compiled into units of execution and data flow relationships among such units. This execution graph relies on a job manager to schedule work to different machines for execution and to provide fault tolerance and recovery, like in MapReduce systems. Each scheduled unit, in turn, can be seen as an independent execution plan and is executed in a runtime environment that borrows ideas from traditional database systems.

The rest of this paper is structured as follows. In Sect. 2, we give a high-level overview of the distributed data platform that supports SCOPE. In Sect. 3, we explain how data are modeled and stored in the system. In Sect. 4, we introduce the SCOPE language. In Sect. 5, we describe in considerable detail the compilation and optimization of SCOPE scripts. In Sect. 6, we introduce the code generation and runtime subsystems, and in Sect. 7, we explain how compiled scripts are scheduled and executed in the cluster. We present a case study in Sect. 8. Finally, we review related work in Sect. 9 and conclude in Sect. 10.

2 Platform architecture

SCOPE relies on a distributed data platform, named Cosmos [7], for storing and analyzing massive data sets. Cosmos is designed to run on large clusters consisting of tens of thousands of commodity servers and has similar goals to other distributed storage systems [3,13]. Disk storage is distributed with each server having one or more direct-attached disks. High-level design objectives for the Cosmos platform include:

Availability: Cosmos is resilient to hardware failures to avoid whole system outages. Data is replicated throughout the system and metadata is managed by a quorum group of servers to tolerate failures.
Reliability: Cosmos recognizes transient hardware conditions to avoid corrupting the system. System components are checksummed end-to-end and the on-disk data is periodically scrubbed to detect corrupt or bit rot data before it is used by the system.
Scalability: Cosmos is designed from the ground up to store and process petabytes of data, and resources are easily increased by adding more servers.
Performance: Data is distributed among tens of thousands of servers. A job is broken down into small units of computation and distributed across a large number of CPUs and storage devices.
Cost: Cosmos is cheaper to build, operate and expand, per gigabyte, than traditional approaches that use smaller number of expensive large-scale servers.

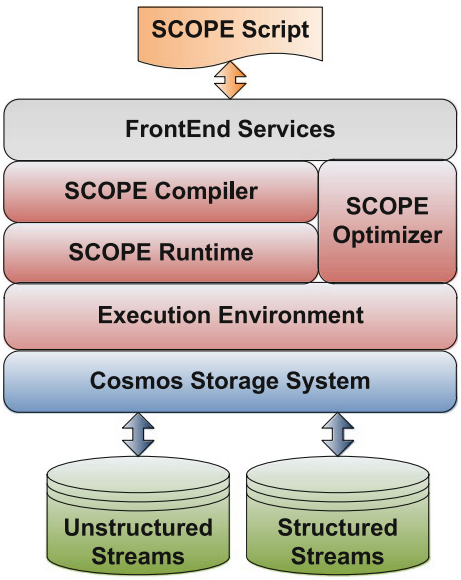

Fig. 1 Architecture of the cosmos platform

Figure 1 shows the architecture of the Cosmos platform. We next describe the main components:

Storage system The storage system is an append-only file system optimized for large sequential I/O. All writes are append-only, and concurrent writers are serialized by the system. Data are distributed and replicated for fault tolerance and compressed to save storage and increase I/O throughput. The storage system provides a directory with a hierarchical namespace and stores sequential files of unlimited size. A file is physically composed of a sequence of *extents*. Extents are the unit of space allocation and replication. They are typically a few hundred megabytes in size. Each computation unit generally consumes a small number of collocated extents. Extents are replicated for reliability and also regularly scrubbed to protect against bit rot. The data within an extent consist of a sequence of *append blocks*. The block boundaries are defined by application appends. Append blocks are typically a few megabytes in size and contain a collection of application-defined records. Append blocks are stored in compressed form with compression and decompression done transparently at the client side. As servers are connected via a high-bandwidth network, the storage system supports both local and remote reads and writes.

Computation system The SCOPE computation system contains the compiler, the optimizer, the runtime, and the execution environment. A query plan is modeled as a dataflow graph: a directed acyclic graph (DAG) with vertices representing processes and edges representing data flows. In the rest of this paper, we discuss these components in detail.

Frontend services This component provides both interfaces for job submission and management, for transferring data in and out of Cosmos for interoperability, and for monitoring job queues, tracking job status and error reporting. Jobs are managed in separate queues, each of which is assigned to a different team with different resource allocations.

3 Data representation

SCOPE supports processing data files in both unstructured and structured formats. We call them *unstructured streams* and *structured streams*, respectively.[1]

3.1 Unstructured streams

Data from the web, such as search logs and click streams, are by nature semi-structured or even unstructured. An unstructured stream is logically a sequence of bytes that is interpreted and understood by users by means of *extractors*. Extractors must specify the schema of the resulting tuples (which allows the SCOPE compiler to bind the schema information to the relational abstract syntax tree) and implement the iterator interface for extracting the data. Analogously, output of scripts (which are rows with a given schema) can be written to unstructured streams by means of *outputters*. Both extractors and outputters are provided by the system for common scenarios and can be supplied by users for specialized situations. Unlike traditional databases, data can be consumed without an explicit and expensive data loading process. SCOPE provides great flexibility to deal with data sources in a variety of formats.

3.2 Structured streams

Structured data can be efficiently stored as structured streams. Like tables in databases, a structured stream has a well-defined schema that every record follows. SCOPE provides an built-in format to store records with different schemas, which allows constant-time access to any column. 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), and access methods. This design makes it possible to understand structured streams and optimize scripts taking advantage of their properties without the need of a separate metadata service.

Partitioning Structured streams can be horizontally partitioned into tens of thousands of partitions. SCOPE supports a variety of partitioning schemes, including hash and range partitioning on a single or composite keys. Based on the data volume and distribution, SCOPE can choose the optimal number of partitions and their boundaries by means of sampling and calculating distributed histograms. Data in a partition are typically processed together (i.e., a partition represents a computation unit). A partition of a structured stream is comprised of one or several physical extents. The approach allows the system to achieve effective replication and fault recovery through extents while providing computation efficiency through partitions.

Data affinity A partition can be processed efficiently when all its extents are stored close to each other. Unlike traditional parallel databases, SCOPE does not require all extents of a partition to be stored on a single machine that could lead to unbalanced storage across machines. Instead, SCOPE attempts to store the extents close together by utilizing *store affinity*. Store affinity aims to achieve maximum data locality without sacrificing uniform data distribution. Every extent has an optional *affinity id*, and all extents with the same *affinity id* belong to an *affinity group*. The system treats store affinity as a placement hint and 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. If the rack is also overloaded, the system then tries to place the extents in a close rack based on the network topology. 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*, so that they are stored close together.

Stream references 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 partitions) 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 less data need not be transferred across the network.

[1] For historical reasons we call data files *streams*, although they are not related to the more traditional concept of read-once streams in the literature.

Indexes for random access Within each partition, a local sort 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 sort keys. Such support is very useful for queries that select only a small portion of the underlying tables and also for more sophisticated strategies such as index-based joins.

Column groups To address scenarios that require processing just a few columns of a wide table, SCOPE supports the notion of *column groups*, which contain vertical partitions of tables over user-defined subsets of columns. As in the Decomposition Storage Model [9], a record-id field (surrogate) is added in each column group so that records can be pieced together if needed.

Physical design Partitioning, sorting, column groups, and stream references are useful design choices that enable efficient execution plans for certain query classes. As in traditional DBMSs, judicious choices based on query workloads could improve query performance by orders of magnitudes. Supporting automated physical design tools in SCOPE is part of our future work.

4 Query language

The SCOPE scripting language resembles SQL but with integrated C# extensions. Its resemblance to SQL reduces the learning curve for users, eases porting of existing SQL scripts into SCOPE, and allows users to focus on application logic rather than dealing with low-level details of distributed system. But the integration with C# also allows users to write custom operators to manipulate row sets where needed. User-defined operators (UDOs) are first-class citizens in SCOPE and optimized in the same way as all other system built-in operators.

A SCOPE script consists of a sequence of commands, which are data manipulation operators that take one or more row sets as input, perform some operation on the data, and output a row set. Every row set has a well-defined schema that all its rows must adhere to. Users can name the output of a command using assignment, and output can be consumed by subsequent commands simply by referring to it by name. Named inputs/outputs enable users to write scripts in multiple (small) steps, a style preferred by some programmers.

In the rest of this section, we describe individual components of the language in more detail.

4.1 Input/output

As explained earlier, SCOPE supports both unstructured and structured streams.

Unstructured streams SCOPE provides two customizable commands, `EXTRACT` and `OUTPUT`, for users to easily read in data from a data source and write out data to a data sink.

Input data to a SCOPE script are extracted by means of built-in or user-defined *extractors*. SCOPE provides standard extractors such as generic text and commonly used log extractors. The syntax for specifying unstructured inputs is as follows:

```
EXTRACT <column>[:<type>] {,<column>[:<type>]}
FROM <stream_name> {,<stream_name>}
USING <extractor> [<args>]
[WITH SAMPLE(<seed>) <number> PERCENT];
```

The `EXTRACT` command extracts data from one or multiple data sources, specified in the `FROM` clause, and outputs a sequence of rows with the schema specified in the `EXTRACT` clause. The optional `WITH SAMPLE` clause allows users to extract samples from the original input files, which is useful for quick experimentation.

SCOPE outputs data by means of built-in or custom *outputters*. The system provides an outputter for text files with custom delimiters and other specialized ones for common tasks. Users can specify expiration dates for streams and thus have some additional control on storage consumption over time. The syntax of the `OUTPUT` command is defined as follows:

```
OUTPUT [<named_rowset>]
TO <stream_name>
[WITH STREAMEXPIRY <timespan>]
[USING <outputter_name> [<output_args>]];
```

Structured streams Users can refer to structured streams as follows:

```
SSTREAM <stream_name>;
```

It is not necessary to specify the columns in structured streams as they are retrieved from the stream metadata during compilation. However, for script readability and maintainability, columns can be explicitly named:

```
SELECT a, b, c FROM SSTREAM <stream_name>;
```

When outputting a structured stream, the user can declare which fields are used for partitioning (`CLUSTERED` clause) and for sorting within partitions (`SORTED` clause). The output can be affinitized to another stream by using the `REFERENCE` clause. The syntax for outputting structured streams is as follows:

```
OUTPUT [<named_rowset>] TO SSTREAM <stream_name>
  [ [HASH | RANGE] CLUSTERED BY <cols>
    [INTO <number>]
    [REFERENCE STREAM <stream_name>]
    [SORTED BY <cols>] ];

cols := <column> [ASC | DESC]
     {,<column [ASC | DESC]}
```

4.2 SQL-like extensions

SCOPE defines the following relational operators: *selection*, *projection*, *join*, *group-by*, *aggregation* and set operators such as *union*, *union-all*, *intersect*, and *difference*. Supported join types include inner joins, all flavors of outer and semi-joins, and cartesian products. SCOPE supports user-defined aggregates and MapReduce extensions, which will be further discussed in the later subsections. The SELECT command in SCOPE is defined as follows:

```
SELECT [DISTINCT] [TOP <count>]
       <select_item> [AS <alias>]
       {,<select_item> [AS <alias>]}
FROM (<named_rowset>|<joined_input>)
     [AS <alias>]
     {,(<named_rowset>|<joined_input>)
     [AS <alias>]}
[WHERE  <predicate>]
[GROUP BY <column> {,<column>}]
[HAVING <predicate>]
[ORDER BY <column> [ASC|DESC]
{,<column> [ASC|DESC]}];

joined_input :=
     <input_stream> <join_type>
     <input_stream>
     [ON <equi_join_predicate>]

join_type := [INNER] JOIN
             | CROSS JOIN
             | [LEFT|RIGHT|FULL] OUTER JOIN
             | [LEFT|RIGHT] SEMI JOIN
```

Nesting of commands in the FROM clause is allowed (i.e., named_rowset can be the result of another command). Subqueries with outer references are not allowed. Nevertheless, most commonly used *correlated subqueries* can still be expressed in SCOPE by combinations of outer-join, semi-join, and user-defined *combiners*.

SCOPE provides several common built-in aggregate functions: COUNT, COUNTIF, MIN, MAX, SUM, AVG, STDEV, VAR, FIRST, and LAST, with optional DISTINCT qualifier. FIRST (LAST) returns the first (last) row in the group (non-deterministically if the rowset is not sorted). COUNTIF takes a predicate and counts only the rows that satisfy the predicate. It is usually used when computing conditional aggregates, such as:

```
SELECT id, COUNTIF(duration<10) AS short,
          COUNTIF(duration>=10) AS long
FROM R
GROUP BY id;
```

The following example shows a very simple script that (1) extracts data in an unstructured stream log.txt containing comma-separated data using an appropriate user-defined extractor, (2) performs a very simple aggregation, and (3) writes the result in an XML file using an appropriate outputter.

```
A = EXTRACT a:int, b:float FROM
    "log.txt" USING CVSExtractor();

B = SELECT a, SUM(b) AS SB
    FROM A
    GROUP BY a;

OUTPUT B TO "log2.xml" USING
XMLOutputter();
```

4.3 .NET integration

In addition to SQL commands, SCOPE also integrates C# into the language. First, it allows user-defined types (UDT) to enrich its type system, so that applications can stay at a higher level of abstraction for dealing with rich data semantics. Second, it allows user-defined operators (UDO) to complement built-in operators. All C# fragments can be either defined in a separate library, or inlined in the script's C# block.

4.3.1 User-defined types (UDT)

SCOPE supports a variety of primitive data types, including bool, int, decimal, double, string, binary, DateTime, and their nullable counterparts. Yet, most of the typical web analysis applications in our environment have to deal with complex data types, such as web documents and search engine result pages. Even though it is possible to shred those complex data types into primitive ones, it results in unnecessarily complex SCOPE scripts to deal with the serialization and deserialization of the complex data. Furthermore, it is common that complex data's internal schemas evolve over time. Whenever that happens, users have to carefully review and revise entire scripts to make sure they will work with newer version of the data. SCOPE UDTs are arbitrary C# classes that can be used as columns in scripts. UDTs provide several benefits. First, applications remain focused on their own requirements, instead of dealing with the shredding of the complex data. Second, as long as UDT interfaces (including methods and properties) remain the same, the applications remain intact despite internal UDT implementation changes. Also, if the UDT schema evolves, the handling of schema versioning and keeping backward compatibility remain inside the implementation of the UDT. The scripts that consume the UDT remain mostly unchanged.

The following example demonstrates the use of a UDT RequestInfo. Objects of type RequestInfo are deserialized into column Request from some binary data on disk using the UDT's constructor. The SELECT statement will then return all the user requests that originated from an *IE* browser by accessing its property Browser and calling the function IsIE() defined by the UDT Browser.

```
SELECT UserId, SessionId,
      new RequestInfo(binaryData)
      AS Request
FROM InputStream
WHERE Request.Browser.IsIE();
```

4.3.2 Scalar UDOs

Scalar UDOs can be divided into scalar expressions, scalar functions, and user-defined aggregates. Scalar expressions are simply C# expressions, and scalar functions take one or more scalar input parameters and return a scalar. SCOPE accepts most of the scalar expressions and scalar functions anywhere in the script. User-defined aggregates must implement the system-defined `Aggregate` interface, which consists of *initialize*, *accumulate*, and *finalize* operations. A user-defined aggregate can be declared *recursive* (i.e., commutative and distributive) in which case it can be evaluated as *local* and *global* aggregation.

4.4 MapReduce-like extensions

For complex data mining and analysis applications, it is sometimes complicated or impossible to express the application logic by SQL-like commands alone. Furthermore, sometimes users have preexisting third-party libraries that are necessary to meet certain data processing needs. In order to accommodate such scenarios, SCOPE provides three extensible commands that manipulate row sets: `PROCESS` (which is equivalent to a Mapper in MapReduce), `REDUCE` (which is equivalent to a Reducer in MapReduce), and `COMBINE` (which generalizes joins and it is not present in MapReduce). These commands complement `SELECT`, which offers easy declarative filtering, joining, arithmetic, and aggregation. Detailed description of these extensions can be found in [7].

Process The `PROCESS` command takes a row set as input, processes each row in turn using the user-defined processor specified in the `USING` clause, and then outputs zero, one, or multiple rows. Processors provide functionality for data transformation in *Extract–Transform–Load* pipelines and other analysis applications. A common usage of processors is to normalize nested data into relational form, a usual first step before further processing. The schema of the output row set is either explicitly specified in the `PRODUCE` clause, or programmatically defined in its interface.

```
PROCESS [<name_rowset>]
USING <processor> [<args>]
[PRODUCE <column> {, <column>}]
[WHERE <predicate>]
[HAVING <predicate>];
```

Processors can accept arguments (`<args>`) that influence their internal behavior. The `WHERE` and `HAVING` clauses are convenient shorthands that can be used to pre-filter the input rows and to post-filter the output rows, respectively, without separate `SELECT` commands.

Reduce The `REDUCE` command provides a way of to implement custom grouping and aggregation. It takes as input a row set that has been grouped by the columns specified in the `ON` clause, processes each group using the reducer specified in the `USING` clause, and outputs zero, one, or multiple rows per group. The reducer function is called once per group. Unlike a user-defined aggregate, it takes a row set and produces a row set, instead of a scalar value. The schema of the output row set is either explicitly specified in the `PRODUCE` clause or defined in the interface of the `Reducer` class. User-defined reducers can be *recursive*, meaning that the reducer function can be applied recursively on partial input groups without changing the final result. Users specify the recursive property of a reducer using annotations and must ensure that the corresponding reducer is semantically correct. For non-recursive reducers, the processing of each group is holistic; therefore, input groups fed to the reducer function are guaranteed to be complete, i.e., containing all rows of the group. Similar to recursive aggregate functions, recursive reducers can be executed as *local* and *global* reduce operations and the optimizer will leverage this information to generate the optimal execution plan.

The syntax of `REDUCE` command is defined as follows. The optional `WHERE` and `HAVING` clauses have the same semantics as in `PROCESS` command. In addition, `REDUCE` allows a `PRESORT` clause, which specifies a given order of tuples on each input group, thereby making it unnecessary to cache and sort data in the reducer itself.

```
REDUCE [ <named_rowset>
[PRESORT <column> [ASC|DESC]
        {,<column> [ASC|DESC]}] ]
ON <column> {,<column>}
USING <reducer> [<args>]
[PRODUCE <column> {,<column>}]
[WHERE <predicate>]
[HAVING <predicate>];
```

Combine The `COMBINE` command provides means for custom join implementations. It takes as inputs two row sets, combines them using the combiner specified in the `USING` clause, and outputs a row set. Both inputs are conceptually divided into groups of rows that share the same value of the join columns (denoted with the `ON` clause). When the combiner function is called, it is passed two matching groups, one from each input. One of the inputs can be empty. The `COMBINE` command thus enables a full outer-join and inside the combiner function, users can implement the actual cus-

```
COMBINE R WITH S ON R.a == S.b
USING MultiSetDifference;

#CS
public class MultiSetDifference : Combiner {
    ...
    public override IEnumerable<Row> Combine(
        RowSet left, RowSet right,
        Row outputRow, string[] args) {
        int rightcount = 0;
        foreach(Row rr in right.Rows)
            rightcount++;
        foreach(Row lr in left.Rows)
            if (--rightcount < 0) {
                lr.Copy(outputRow);
                yield return outputRow;
            }
    }
}
#ENDCS
```

Fig. 2 Example of a user-defined combiner

tom join logic. Non-equality predicates can be expressed as residual predicates in the HAVING clause. The optional PRESORT and PRODUCE are the same as for REDUCE commands.

```
COMBINE <named_input1> [AS <alias1>]
   <presort_clause>
   WITH <named_input2> [AS <alias2>]
   <presort_clause>
ON <column>=<column> {AND <column>=
<column>}
USING <combiner> [<args>]
[PRODUCE <column> {, <column>}]
[HAVING <predicate>];
```

Figure 2 shows a simple combiner the computes the differences between two multisets.

4.4.1 Code properties and data properties

In addition to runtime and compile-time interfaces, processors, reducers, and combiners implement optional properties that can be used during optimization.

For any output column, column dependencies describe the set of input columns that this output column functionally depends on. A special case of column dependencies is *pass-through* columns. A column of an output row is defined as pass-through, if the value of the column in that output row is simply a copy of the value from the input row. Pass-through columns allow optimizer to reason about *structural properties* (defined in Sect. 5) between operator inputs and outputs. In Sect. 5, we show how the optimizer leverages the above properties during optimization.

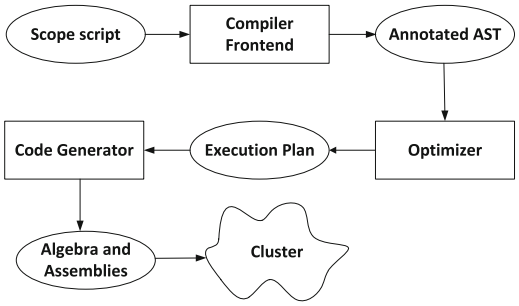

Fig. 3 Compilation of a SCOPE script

4.5 Other language components

Due to space constraints, we cannot cover all features of the language. Nevertheless, the following language features are worth mentioning as they are commonly used in our environment.

- Views: SCOPE allows users to define views as any combination of SCOPE commands with a single output. The schema of the output will be the schema of the view. Views provide a mechanism for data owners to define a higher level of abstraction for the data usage without exposing the implementation detailed organization of the underlying data.
- Macros: SCOPE incorporates a rich macro preprocessor, which is useful for conditional compilation based on input parameters to the script or view.
- Streamsets: Users can refer to multiple streams simultaneously using parameterized names. This is useful for daily streams that need to be processed together in a script. For instance, the following script fragment operates over 10 streams:

```
EXTRACT a, b FROM "log_%n.txt?n=1...10"
USING DefaultTextExtractor;
```

5 Query compilation and optimization

A SCOPE script goes through a series of transformations before it is executed in the cluster, as shown in Fig. 3. 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. The optimizer returns an execution plan that specifies the steps that are required to efficiently execute the script. Finally, code generation produces the final algebra (which details the units of execution and data dependencies among them) and the assemblies that contain user-defined code. This package is then sent to the cluster, where it is actually executed. In the rest of this section, we take a close look at the SCOPE query optimizer [32].

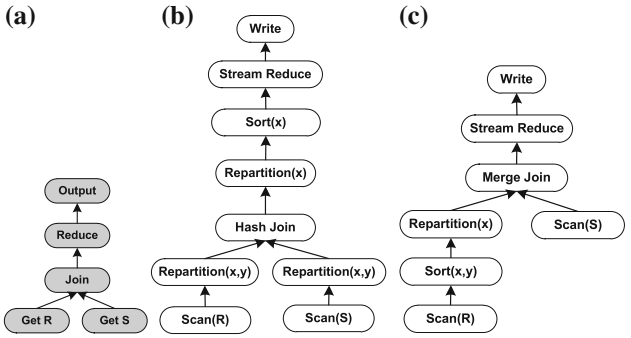

Fig. 4 Input and output trees during query optimization. **a** Logical input tree, **b** non-optimal physical tree, **c** optimal physical tree

Algorithm 1: OptimizeExpr($expr$, $reqd$)

```
Input: Expression expr, ReqdProperties reqd
Output: QueryPlan plan
/* Enumerate all the logical rewrites     */
LogicalTranform(expr);
foreach logical expression lexpr do
    /* Try implementations for root operator
       */
    PhysicalTranform(lexpr);
    foreach expression pexpr that has physical
    implementation for its root operator do
        ReqdProperties reqdChild =
        DetermineChildReqdProperties(pexpr, reqd);
        /* Optimize child expressions        */
        QueryPlan planChild =
        OptimizeExpr(pexpr.Child, reqdChild);
        DlvdProperties dlvd =
        DeriveDlvdProperties(planChild);
        if PropertyMatch(dlvd, reqd) then
            EnqueueToValidPlans();
        end
    end
end
plan = CheapestQueryPlan();
return plan;
```

5.1 Optimizer overview

SCOPE uses a transformation-based optimizer based on the Cascades framework [15] that translates input scripts into efficient execution plans. In addition to traditional optimization techniques, the SCOPE optimizer reasons about partitioning, grouping, and sorting properties in a single uniform framework, and seamlessly generates and optimizes both serial and parallel query plans. Consider, as an example, the simple script

```
R = SSTREAM "R.ss";
S = SSTREAM "S.ss";
J = SELECT * FROM R, S WHERE R.x=S.x AND
    R.y=S.y;
O = REDUCE J ON x USING SampleReducer;
OUTPUT O TO SSTREAM "O.ss";
```

where `S.ss` was generated by the following command (partitioned by `x` and sorted by `x,y`):

```
OUTPUT TO "S.ss" CLUSTERED BY x SORTED BY x,y;
```

Figure 4a shows a tree of logical operators that specifies, in an almost one-to-one correspondence, the relational algebra representation of the script above. Figure 4b, c show two different physical operator trees corresponding to alternative execution plans for the same script.

We illustrate the importance of script optimization by contrasting Fig. 4b, c. In the straightforward execution plan in Fig. 4b, table R is first hash-partitioned on columns $\{R.x, R.y\}$. This step is done in parallel, with each machine processing its portion of the table. Similarly, table S is partitioned on $\{S.x, S.y\}$. Next, rows from matching R and S partitions are hash-joined in parallel, producing a result partitioned on $\{R.x, R.y\}$ (or $\{S.x, S.y\}$). In preparation for the subsequent reducer, the join result is first partitioned by x and then locally sorted by x, to ensure that tuples with the same value of x are in the same partition, sorted by x. Finally, each partition is reduced in parallel, producing the final result.

This plan is expensive because it contains multiple partitioning operations. If both R and S contain tens of terabytes of data, data reshuffling through the network can pose a serious performance bottleneck. The alternative plan in Fig. 4c shows a different plan where some repartition operators have been eliminated. Note that the join operator requires, for correctness, that no two tuples from R (also from S) share the same (x, y) values but belong to different partitions. This can be achieved by repartitioning both inputs on any non-empty subset of $\{x, y\}$. The plan in Fig. 4c repartitions data on x alone and thus (1) avoids shuffling data from S by leveraging existing partitioning and sorting properties and (2) avoids a subsequent repartitioning on x, required by the reducer operator. It also uses a merge join implementation, which requires the inputs to be sorted by (x, y). This is done by sorting R and obtained for free in the case of S due to a preexisting order. As the merge join preserves the order of results in x, no additional sort is required before the reducer.

The SCOPE optimizer considers these (and many other) alternative plans and chooses the one with the lowest estimated costs, based on data statistics and an internal cost model. The cost model of the optimizer is similar to that of traditional database systems. Each operator computes its own cost (including CPU, I/O, and network utilization), and local costs are aggregated to obtain the estimated cost of an execution plan. In this paper, we focus on how the SCOPE optimizer distinguishes itself in reasoning and optimizing parallel query plans and omit other aspects due to space limitations.

Transformation-based optimization can be viewed as divided into two phases, namely logical exploration and

physical optimization. Logical exploration applies transformation rules that generate new logical expressions. Implementation rules, in turn, convert logical operators to physical operators. Algorithm 1 shows a (simplified) recursive optimization routine that takes as input a query expression and a set of requirements. We highlight three different contexts where reasoning about data properties occurs during query optimization.

- *Determining child required properties.* The parent (physical) operator imposes requirements that the output from the current physical operator must satisfy. For example, the output must be sorted on $R.y$. To function correctly, the operator may itself impose certain requirements on its inputs, for example, the two inputs to a join must be partitioned on $R.x$ and $S.x$, respectively. Based on these two requirements, we must then determine what requirements to impose on the result of the input expressions. The function `DetermineChildReqdProperties` is used for this purpose. If the requirements are incompatible, a compensating operator (e.g., sort or partition) would be added during the optimization of the child operator by an enforcer rule (see Sect. 5.5.1).
- *Deriving delivered properties.* Once physical plans for the child expressions have been determined, we compute the data properties of the result of the current physical operator by calling the function `DeriveDlvdProperties`. A child expression may not deliver exactly the requested properties. For example, we may have requested a result grouped on $R.x$ but the chosen plan delivers a result that is, in addition, sorted on $R.x$. The delivered properties are a function of the delivered properties of the inputs and the behavior of the current operator, for example, whether it is hash or merge join. We explain this process in Sect. 5.4.1.
- *Property matching.* Once the delivered properties have been determined, we test whether they satisfy the required properties, by calling the function `PropertyMatch`. If they do not match, the plan with the current operator is discarded. The match does not have to be exact – a result with properties that exceed the requirements is acceptable. We cover the details in Sect. 5.4.3.

We next discuss some key operators in the SCOPE optimizer in Sect. 5.2, our property formalism in Sect. 5.3, reasoning about data properties in Sect. 5.4, and domain-specific transformation rules in Sect. 5.5.

5.2 Operators and parallel plans

The SCOPE optimizer handles all traditional logical and physical operators from relational DBMSs, such as join and union variants, filters, and aggregates. It is also enhanced with specialized operators that correspond to the user-defined operators described in Sect. 4.4 (i.e., extractors, reducers, processors, combiners, and outputters). We next describe two operators that allow the optimizer to reuse common subexpressions and to consider parallel plans in an integrated manner.

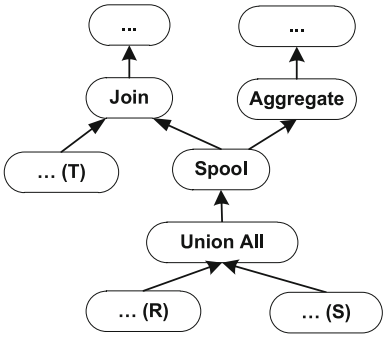

Fig. 5 Common subexpressions during optimization

5.2.1 Common subexpressions

As explained in Sect. 4, SCOPE allows programmers to write scripts as a series of simple data transformations. Due to complex business logics, it is common that scripts explicitly share common subexpressions. As an example, the following script fragment unions two sources into an intermediate result, which is both aggregated and joined with a different source, producing multiple results that are later consumed in the script:

```
...
U = SELECT * FROM R UNION ALL SELECT * FROM S;
G = SELECT a, SUM(b) FROM U GROUP BY a;
J = SELECT * FROM U, T ON U.a = T.a AND
    U.b = T.b;
...
```

The optimizer represents common subexpressions by *spool* operators. Figure 5 shows the logical input to the optimizer for the fragment above. It is important to note that spool operators generalize input and output trees to DAGs. A spool operators has a single producer (or child) and multiple consumers (or ancestors) in the operator DAG. Detailed common subexpression optimization with spool operators can be found in [33]. We also note that sharing expressed at the logical level does not *necessarily* translate into sharing in physical plans. The reason is subtle: it might be more beneficial to execute the shared subexpression twice (each one with different partitioning or sorting properties).

5.2.2 Data exchange

A key feature of distributed query processing is based on partitioning data into smaller subsets and processing partitions

in parallel on multiple machines. This can done by a single operator, the *data exchange* operator, which repartitions data from n inputs to m outputs [14].

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, and m merge operators, one on each output machine. A partition operator reads its input and splits it onto m subpartitions. Each merge operator collects the data for its partition from the n corresponding subpartitions.

Exchange topology Figure 6 shows the three main types of exchange operators. *Initial Partitioning*, shown in Fig. 6a, consumes a single input stream and outputs m output streams with the data partitioned among the m streams. *(Full) Repartitioning*, shown in Fig. 6b, consumes n input partitions and produces m output partitions, partitioned in a different way. *Full Merge*, shown in Fig. 6c, consumes n input streams and merges then into a single output stream. *Partial Partitioning*, shown in Fig. 6d, takes n input streams and produces kn output streams. The data from each input partition are further partitioned among k output streams. Finally, *Partial Merge*, shown in Fig. 6e, is the inverse of partial partition. A partial merge takes kn input streams, merges groups of k of them together, and produces n output streams.

Partitioning schemes Conceptually, 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. We assume all partition operators are FIFO (first-in and first-out), so the order of two rows r_1 and r_2 in the input stream is preserved in 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 schemes include *Nondeterministic Partitioning*, which is any scheme where the data content of a row does not affect which partition the row is assigned to (e.g., round-robin and random), and *Broadcasting*, which outputs a copy of each row to every partition (so that every output stream is a copy of the input stream).

Merging schemes 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. The output stream iteratively consumes the smallest element at the head of the input streams. If the input streams are sorted by the same columns as the parameter of sort-merge, the output is fully sorted. *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 the first row 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.

5.3 Property formalism

In a distributed system, parallelism is achieved by partitioning data into subsets that can be processed independently. This may require complete repartitioning, which is expensive because it involves transporting all data across the shared network. Reducing the number of partitioning operations is an important optimization goal. However, data partitioning cannot be considered in isolation because it often interacts with other data properties like sorting and grouping. Reasoning about partitioning, grouping, and sorting properties, and their interactions, is an important

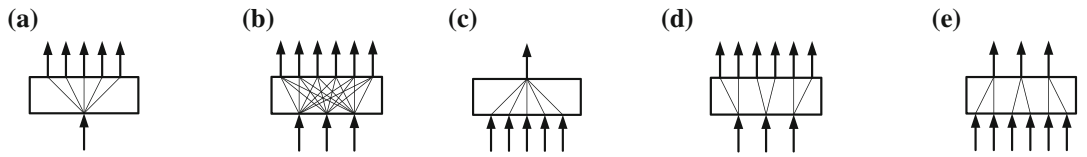

Fig. 6 Different types of data exchange. **a** Initial partitioning, **b** repartitioning, **c** full merge, **d** partial repartitioning, **e** partial merge

Table 1 Notation used in the paper

$C_1, C_2, \ldots$	Columns
$\mathcal{X}, \mathcal{Y}, \mathcal{S}, \mathcal{G}, \mathcal{J}$	Sets of columns
$\mathcal{A}, \mathcal{B}$	Local structural properties
$r_1, r_2, \ldots$	Tuples
$\mathrm{P}_1, \mathrm{P}_2, \ldots$	Partitions
$r[C], r[\mathcal{X}]$	Projection of r onto column C and columns $\mathcal{X}$, respectively
$*$	Any properties (including empty)
$\perp, \emptyset, \top$	Non-partitioned, randomly partitioned, and replicated global properties

foundation for query optimization in a distributed environment.

A partition operation divides a relation into disjoint subsets, called partitions. A partition function defines which rows belong to which partitions. Partitioning applies to the whole relation; it is a *global structural property*. Grouping and sorting properties define how the data within each partition is organized and are thus partition-local properties, here refereed to as *local structural properties*. We next define these three properties, collectively referred to as *structural (data) properties*. Table 1 summarizes the notation used in this section.

5.3.1 Local structural properties

A sequence of rows $r_1, r_2, \ldots, r_m$ is *grouped* on a set of columns $\mathcal{X}$, denoted $\mathcal{X}^g$, if between two rows that agree in $\mathcal{X}$, there is no row with a different value in $\mathcal{X}$. That is, if $\forall r_i, r_j : i < j \wedge r_i[\mathcal{X}] = r_j[\mathcal{X}] \Rightarrow \forall k : i < k < j, r_k[\mathcal{X}] = r_i[\mathcal{X}]$. Similarly, a sequence of rows $r_1, r_2, \ldots, r_m$ is *sorted* on a column C in an ascending (or descending) order, denoted by $C^{o\uparrow}$ (or $C^{o\downarrow}$), if $\forall r_i, r_j, i < j \Rightarrow r_i[C] \leq r_j[C]$ (or $r_i[C] \geq r_j[C]$). We use C^o for simplicity when the context is clear.

Definition 1 (*Local structural properties*) Local structural properties $\mathcal{A}$ are ordered sequences of grouping and sorting properties $\mathcal{A} = (A_1, A_2, \ldots, A_n)$, where each A_i is either $\mathcal{X}^g$ or C^o. A sequence of rows $R = (r_1, r_2, \ldots, r_m)$ satisfies local structural properties $\mathcal{A}$ if (1) R satisfies $(A_1, A_2, \ldots, A_{n-1})$, and (2) every subsequence of R that agrees on the values of columns in $A_1, A_2, \ldots, A_{n-1}$ additionally satisfies A_n.

We denote $(A_0 : \mathcal{A})$ to be the concatenation of a grouping or sorting property A_0 to local structural properties $\mathcal{A}$, that is, $(A_0, A_1, A_2, \ldots, A_n)$.

5.3.2 Global structural properties

There are two major classes of partitioning schemes, *ordered* and *non-ordered*. A non-ordered partitioning scheme ensures only that all rows with the same values of the partitioning columns are contained in the same partition. This is analogous to *grouping* as local property. More formally, a relation $\mathcal{R}$ is *non-ordered partitioned* on columns $\mathcal{X}$ with partitioning function P if $\forall r_1, r_2 \in \mathcal{R} : r_1[\mathcal{X}] = r_2[\mathcal{X}] \Rightarrow \mathrm{P}(r_1) = \mathrm{P}(r_2)$.

An ordered partitioning scheme provides the additional guarantee that the partitions cover disjoint ranges of the partitioning columns. In other words, rows assigned to a partition P_i are either all less than or greater than rows in another partition P_j. This is analogous to *ordering* as a local property. More formally, a relation $\mathcal{R}$ is *ordered-partitioned* into partitions $\mathrm{P}_1, \mathrm{P}_2, \ldots, \mathrm{P}_m$ on $(C_1^{o_1}, C_2^{o_2}, \ldots, C_n^{o_n})$ where $o_i \in \{o_\uparrow, o_\downarrow\}$, if it satisfies the non-ordered partitioned condition above and, for each pair of partitions P_i and P_j with $i < j$:

$$\forall r_i \in \mathrm{P}_i, r_j \in \mathrm{P}_j : r_i[C_1, \ldots, C_n] <_{o_1,\ldots,o_n} r_j[C_1, \ldots, C_n]$$

where $<_{o_1,\ldots,o_n}$ is the multi-column comparison operator that takes ascending/descending variants into account, so that for instance $(1, 2, 3) <_{o_\uparrow, o_\downarrow, o_\uparrow} (1, 5, 2)$.

Definition 2 (*Global structural properties*) Given a relation $\mathcal{R}$ and columns $\mathcal{X} = \{C_1, \ldots C_n\}$, a global structural property can be (1) $\perp$, which indicates that data are not partitioned, (2) $\emptyset$, which indicates that data are randomly partitioned, (3) $\top$, which indicates that data are completely duplicated, (4) $\mathcal{X}^g$, which indicates that $\mathcal{R}$ is non-ordered partitioned on columns $\mathcal{X}$, and (5) $(C_1^{o_1}, \ldots, C_n^{o_n})$, which indicates that $\mathcal{R}$ is ordered partitioned on $(C_1^{o_1}, \ldots, C_n^{o_n})$. When it is clear from the context, we use $(C_1^{o_1}, \ldots, C_n^{o_n})$, and $\mathcal{X}^o$ interchangeably for an ordered partitioned global property.

5.3.3 Structural properties

The structural properties of a relation $\mathcal{R}$ are specified by a pair of global structural property $\mathbb{G}$ and local structural properties $\mathbb{L}$, denoted as $\{\mathbb{G}; \mathbb{L}\}$. Additionally, a structural property also defines the exact partitioning function and the number of partitions, which we omit in this work for the clarity of exposition.

Table 2 shows an instance of a relation with three columns $\{C_1, C_2, C_3\}$, and structural data properties $\{\{C_1\}^g; (\{C_1, C_2\}^g, C_3^o)\}$. In words, the relation is partitioned on column C_1 and, within each partition, data are first grouped on columns C_1, C_2, and, within each such group, sorted by column C_3.

Table 2 Relation with partitioning, grouping, and sorting

Partition 1	Partition 2	Partition 3
{1,4,2}, {1,4,5}, {7,1,2}	{4,1,5}, {3,7,8}, {3,7,9}	{6,2,1}, {6,2,9}

5.4 Structural properties and query optimization

We next describe how the optimizer reasons about data properties in the three contexts outlined in Algorithm 1.

5.4.1 Deriving structural properties

We first consider how to derive the structural properties of the output of a *physical* operator. Earlier research has shown how to derive ordering and grouping properties for standard relational operators executed on non-partitioned inputs [21, 22,25,26,30]. Ordering and grouping are local properties, that is, properties of each partition, so previous work still applies when the operators are running in partitioned mode. What remains is to reason with global partitioning properties throughout a query plan and their interaction with local properties.

Properties after a scan operator Recall from Sect. 3 that data can be stored either as unstructured or structured streams. The scan operator over an unstructured stream delivers a random partitioning. More interestingly, reading a structured stream delivers the partitioning and sorting properties that are defined in the structured stream itself. This can be useful to avoid further repartitioning operations whenever the required properties of a subsequent operator can be satisfied by the delivered properties of the structured stream itself.

Properties after a partitioning operator Partition operators are assumed to be FIFO, that is, they output rows in the same order that they are read from the input. Thus, they affect the global properties but not local properties. Every output partition inherits the local properties (sorting and grouping) of its input. Table 3 summarizes the properties of the output after an initial partitioning operator when the input has structural properties $\{\mathcal{X}^\theta; \mathcal{A}\}$. Hash partitioning on columns $C_1, C_2, \ldots, C_n$ produces a non-ordered collection of partitions, which is indicated in the table with the global structural property $\{C_1, C_2, \ldots, C_n\}^g$. Range partitioning on columns $C_1, C_2, \ldots, C_n$ produces an ordered collection of partitions, which is indicated in the table with global structural property $(C_1^{o_1}, C_2^{o_2}, \ldots, C_n^{o_n})$. In a non-deterministic partitioning scheme (round-robin and random partitioning), a row is partitioned independent on its content, so we indicate this by $\emptyset$.

Table 3 Structural properties of the result after partitioning an input with properties $\{\mathcal{X}; \mathcal{A}\}$

Scheme	Result
Hash on $C_1, \ldots, C_n$	$\{\{C_1, \ldots, C_n\}^g; \mathcal{A}\}$
Range on $C_1^{o_1}, \ldots, C_n^{o_n}$	$\{(C_1^{o_1}, \ldots, C_n^{o_n}); \mathcal{A}\}$
Non-deterministic	$\{\emptyset; \mathcal{A}\}$
Broadcast	$\{\top; \mathcal{A}\}$

Properties after a merge operator A full merge operator produces a single output. Its local properties depend on the local properties of the input and the merge operator type: random merge, sort-merge, concat merge, and sort-concat merge.

Table 4 summarizes the structural properties after a full merge, depending on the type of merge operator and whether the input partitioning is ordered or non-ordered. A random merge does not guarantee any row order in the result, so no local properties can be derived for the output. For a sort-merge, there are two cases. If the local properties of the input imply that the input streams are sorted on the columns used in the merge (see Sect. 5.4.3 for more details on property inference), the output will be sorted, otherwise not. A concat merge operator maintains the row order within each source partition. If each source partition is *grouped* in a similar way to how it is *non-ordered partitioned*, the result of is also grouped, otherwise not. Finally, a sort-concat merge produces a sorted result if inputs are range partitioned and each partition is also sorted on the same columns as it is partitioned on.

Example 1 A sort-concat full merge on $\{C_1^o, C_2^o\}$ of inputs with properties $\{(C_1^o, C_2^o); (C_1^o, C_2^o, C_3^g)\}$ generates an output with properties $\{\perp; (C_1^o, C_2^o, C_3^g)\}$.

Properties after a repartitioning operator The properties of the result after repartitioning depend on the partitioning scheme, the merge scheme, and the local properties of the input. Table 5 summarizes the structural properties after repartitioning an input with properties $\{*; \mathcal{A}\}$.

Example 2 Given inputs with properties $\{\{C_1, C_2\}^g; (C_2^o, C_1^o, C_3^o)\}$, concat merging generates an output with properties $\{\{C_1, C_2\}^g; (\{C_1, C_2\}^g, C_3^o)\}$, for repartitioning, and $\{\perp; (\{C_1, C_2\}^g, C_3^o)\}$, for a full merge.

5.4.2 Deriving required structural properties

We now consider how to determine required properties of the inputs for different physical operators. Table 6 lists required input properties for the most common physical operators.

Table 4 Structural properties of the result after a full merge

	Input properties $\{\mathcal{X}^g; \mathcal{A}\}$	Input properties $\{\mathcal{X}^o; \mathcal{A}\}$
Random merge	$\{\perp; \emptyset\}$	$\{\perp; \emptyset\}$
Sort merge on $\mathcal{S}^o$	(1). $\{\perp; \mathcal{S}^o\}$ if $\mathcal{A} \Rightarrow \mathcal{S}^o$	(1). $\{\perp; \mathcal{S}^o\}$ if $\mathcal{A} \Rightarrow \mathcal{S}^o$
	(2). $\{\perp; \emptyset\}$ otherwise	(2). $\{\perp; \emptyset\}$ otherwise
Concat merge	(1). $\{\perp; (\mathcal{X}^g : \mathcal{B})\}$ if $\mathcal{A} \Rightarrow (\mathcal{X}^g : \mathcal{B})$	(1). $\{\perp; (\mathcal{X}^g : \mathcal{B})\}$ if $\mathcal{A} \Rightarrow (\mathcal{X}^g : \mathcal{B})$
	(2). $\{\perp; \emptyset\}$ otherwise	(2). $\{\perp; \emptyset\}$ otherwise
Sort-concat merge on $\mathcal{S}^o$	(1). $\{\perp; (\mathcal{X}^g : \mathcal{B})\}$ if $\mathcal{A} \Rightarrow (\mathcal{X}^g : \mathcal{B})$	(1). $\{\perp; \mathcal{A}\}$ if $\mathcal{S}^o \Leftrightarrow \mathcal{X}^o$ and $\mathcal{A} \Rightarrow \mathcal{S}^o$
	(2). $\{\perp; \emptyset\}$ otherwise	(2). $\{\perp; \emptyset\}$ otherwise

Table 5 Structural properties of the result after repartitioning on $\mathcal{X}$ with input properties $\{*; \mathcal{A}\}$

	Hash partitioning	Range partitioning	Non-determ. partitioning
Random merge	$\{\mathcal{X}^g; \emptyset\}$	$\{\mathcal{X}^o; \emptyset\}$	$\{\emptyset; \emptyset\}$
Sort merge on $\mathcal{S}^o$	$\{\mathcal{X}^g; \mathcal{S}^o\}$ if $\mathcal{A} \Rightarrow \mathcal{S}^o$	$\{\mathcal{X}^o; \mathcal{S}^o\}$ if $\mathcal{A} \Rightarrow \mathcal{S}^o$	$\{\emptyset; \mathcal{S}^o\}$ if $\mathcal{A} \Rightarrow \mathcal{S}^o$
	$\{\mathcal{X}^g; \emptyset\}$ otherwise	$\{\mathcal{X}^o; \emptyset\}$ otherwise	$\{\emptyset; \emptyset\}$ otherwise
Concat merge	$\{\mathcal{X}^g; (\mathcal{X}^g : \mathcal{B})\}$ if $\mathcal{A} \Rightarrow (\mathcal{X}^g : \mathcal{B})$	$\{\mathcal{X}^o; (\mathcal{X}^g : \mathcal{B})\}$ if $\mathcal{A} \Rightarrow (\mathcal{X}^g : \mathcal{B})$	$\{\emptyset; (\mathcal{X}^g : \mathcal{B})\}$ if $\mathcal{A} \Rightarrow (\mathcal{X}^g : \mathcal{B})$
	$\{\mathcal{X}^g; \emptyset\}$ otherwise	$\{\mathcal{X}^o; \emptyset\}$ otherwise	$\{\emptyset; \emptyset\}$ otherwise
Sort-concat merge on $\mathcal{S}^o$	$\{\mathcal{X}^g; (\mathcal{X}^g : \mathcal{B})\}$ if $\mathcal{A} \Rightarrow (\mathcal{X}^g : \mathcal{B})$	$\{\mathcal{X}^o; \mathcal{A}\}$ if $\mathcal{S}^o \Leftrightarrow \mathcal{X}^o$ and $\mathcal{A} \Rightarrow \mathcal{S}^o$	$\{\emptyset; \mathcal{A}\}$ if $\mathcal{S}^o \Leftrightarrow \mathcal{X}^o$ and $\mathcal{A} \Rightarrow \mathcal{S}^o$
	$\{\mathcal{X}^g; \emptyset\}$ otherwise	$\{\mathcal{X}^o; \emptyset\}$ otherwise	$\{\emptyset; \emptyset\}$ otherwise

Table 6 Required structural properties of inputs to common physical operators

	Non-partitioned version	Partitioned version
Table scan, select, project	$\{\perp; *\}$	$\{\mathcal{X}^\theta; *\}, \mathcal{X} \neq \emptyset$
Hash aggregate on $\mathcal{G}$	$\{\perp; *\}$	$\{\mathcal{X}^\theta; *\}, \emptyset \subset \mathcal{X} \subseteq \mathcal{G}$
Stream aggregate on $\mathcal{G}$	$\{\perp; (\mathcal{G}^g)\}$	$\{\mathcal{X}^\theta; (\mathcal{G}^g)\}, \emptyset \subset \mathcal{X} \subseteq \mathcal{G}$
Nested-loop or hash join (equijoin on columns $\mathcal{J}_1 \equiv \mathcal{J}_2$)	Both inputs $\{\perp; *\}$	*Pair-wise Join:* Input 1: $\{\mathcal{X}^\theta; *\}, \emptyset \subset \mathcal{X} \subseteq \mathcal{J}_1$; Input 2: $\{\mathcal{Y}^\theta; *\}, \emptyset \subset \mathcal{Y} \subseteq \mathcal{J}_2; \mathcal{X} \equiv \mathcal{Y}$ *Broadcast Join:* Input 1: $\{\top; *\}$; Input 2: $\{\mathcal{X}^\theta; *\}, \mathcal{X} \neq \emptyset$
Merge join (equijoin on columns $\mathcal{J}_1 \equiv \mathcal{J}_2$)	Input 1: $\{\perp; \mathcal{J}_1^o\}$ Input 2: $\{\perp; \mathcal{J}_2^o\}$	*Pair-wise Join:* Input 1: $\{\mathcal{X}^\theta; \mathcal{J}_1^o\}, \emptyset \subset \mathcal{X} \subseteq \mathcal{J}_1$; Input 2: $\{\mathcal{Y}^\theta; \mathcal{J}_2^o\}, \emptyset \subset \mathcal{Y} \subseteq \mathcal{J}_2; \mathcal{X} \equiv \mathcal{Y}$ *Broadcast Join:* Input 1: $\{\top; \mathcal{J}_1^o\}$; Input 2: $\{\mathcal{Y}; \mathcal{J}_2^o\}, \mathcal{Y} \neq \emptyset$

Note that for the purpose of property derivation, we treat combiners and reducers as joins and aggregates, respectively. Depending on whether the operator is executed in either partitioned or non-partitioned mode, it imposes different requirements on its inputs. A non-partitioned operator runs as a single instance and produces non-partitioned data. A partitioned operator runs as multiple instances, each instance consuming and producing only partitions of the data. However, not all operators can be executed in parallel. For instance, aggregation with no group-by columns can only be executed in non-partitioned mode.

Table scan, select and project process individual rows and impose no requirements on their inputs (i.e., it does not matter how the input data are partitioned, sorted, or grouped). Thus, their input requirements are shown as $\{\mathcal{X}^\theta; *\}$ where $\mathcal{X}$ can be any set of columns.

For a hash aggregation to work correctly, all rows with the same value of the grouping columns must be in a single partition. This is guaranteed as long as the input is partitioned on a *subset* of the grouping columns. A stream aggregation also requires that the rows within each partition be grouped on the grouping columns.

We consider two types of partitioned joins: pair-wise join and broadcast join. A pair-wise join takes two partitioned inputs. The inputs must be partitioned on a subset of the join columns in the same way (i.e., on the same set of equivalent

columns and into the same number of partitions). Broadcast join takes one partitioned input (it does not really matter how it is partitioned) and a second input that is replicated (broadcast) to each partition of the first input. A merge join has the additional requirement that each partition be sorted on the join columns.

Example 3 Suppose, we are considering using a partitioned merge join to join tables R and S on $R.C_1 = S.C_1$ and $R.C_2 = S.C_2$. Based on the rules in Table 6, both inputs must be partitioned and sorted in the same way. The partitioning columns must be a subset of or equal to the join columns ($\{R.C_1, R.C_2\}$ and $\{S.C_1, S.C_2\}$, respectively). The sort columns must also be equal to the join columns on each input. Each of the following requirements satisfies the restrictions and is thus valid input requirements. We do not list all possibilities here and also leave the exact sort order, $o_\uparrow$ and $o_\downarrow$, unspecified.

- $\{R.C_1^g; (R.C_2^o, R.C_1^o)\}$ and $\{S.C_1^g; (S.C_2^o, S.C_1^o)\}$
- $\{R.C_2^o; (R.C_1^o, R.C_2^o)\}$ and $\{S.C_2^o; (S.C_1^o, S.C_2^o)\}$
- $\{\{R.C_1, R.C_2\}^g; (R.C_2^o, R.C_1^o)\}$ and $\{\{S.C_1, S.C_2\}^g; (S.C_2^o, S.C_1^o)\}$

As shown by the example, the requirements in Table 6 for the child expressions are not always unique and can be satisfied in several ways. For instance, aggregation on $\{C_1, C_2\}$ requires the input to be partitioned on $\{C_1\}$, $\{C_2\}$, or $\{C_1, C_2\}$. Conceptually, each requirement corresponds to one specific implementation. This situation could be handled by generating multiple alternatives, one for each requirement. However, this approach would generate a large number of alternatives, making optimization more expensive. Instead, we allow required properties to cover a *range* of possibilities and rely on enforcer rules, described in Sect. 5.5.1, to generate valid rewrites. To this end, the optimizer encodes required structural properties as follows.

- Partitioning requirement, which can be either broadcast ($\top$), non-partitioned ($\bot$), or a partition requirement including *minimum* partitioning columns $\mathcal{X}_{\min}$ and *maximum* partitioning columns $\mathcal{X}_{\max}$ ($\emptyset \subseteq \mathcal{X}_{\min} \subseteq \mathcal{X}_{\max}$).
- Sorting requirement, which consists of a sequence of sorting columns $(S_1^o, S_2^o, \ldots, S_n^o)$, $o \in \{o_\uparrow, o_\downarrow\}$.
- Grouping requirement, which consist of a set of grouping columns $\{G_1, G_2, \ldots, G_n\}^g$.

In the previous example of aggregation on $\{C_1, C_2\}$, the partitioning requirement is any subset of the grouping columns, so $\mathcal{X}_{\min} = \emptyset, \mathcal{X}_{\max} = \{C_1, C_2\}$. This requirement is satisfied by a hash or range partition with column set $\mathcal{X}$ where $\mathcal{X}_{\min} \subset \mathcal{X} \subseteq \mathcal{X}_{\max}$.

Some operators are by themselves invariant to partitioning and ordering (e.g., filters) and simply pass parent requirements down to their children.

User-defined operators deserve a special mention. In principle, a `Processor` can execute arbitrary code to produce output rows given its input. Therefore, it would seem that any property inference cannot propagate past such operators. The properties described in Sect. 4.4.1 provide a way for users to specify additional information to the optimizer. For instance, if a processor marks a column as pass-through, it essentially means that input and output rows would satisfy all structural properties on such column. Therefore, we can push down a partitioning requirement through a processor as for the case of filters whenever all the structural property columns are marked as pass-through. These and other properties (e.g., column dependencies) are also used to determine the required output columns from a child operator, essentially performing a guided data-flow analysis on the query DAG.

The rules in Table 6 do not consider requirements imposed on the operator by its parent. In that case, some operators are able to do early pruning due to conflicting properties. For instance, if a merge join is required to produce a result sorted on (C_1, C_2) but its equality join predicates are on C_3, there is no merge join implementation that could satisfy its sorting requirements, assuming that sorting on (C_3) does not imply sorting on (C_1, C_2). This merge join is an invalid alternative—it can never produce an output that satisfies the requirements. The optimizer checks for such invalid alternatives and discards them immediately.

5.4.3 Property matching

The optimizer ensures that a physical plan is valid by checking that its delivered properties satisfy the required properties. Property matching checks whether one set of properties $\{\mathcal{X}^{\theta_1}; \mathcal{A}\}$ satisfies another $\{\mathcal{Y}^{\theta_2}; \mathcal{B}\}$, that is, whether $\{\mathcal{X}^{\theta_1}; \mathcal{A}\} \Rightarrow \{\mathcal{Y}^{\theta_2}; \mathcal{B}\}$. Matching of structural properties is done by matching global and local properties separately:

$$\left(\{\mathcal{X}^{\theta_1}; \mathcal{A}\} \Rightarrow \{\mathcal{Y}^{\theta_2}; \mathcal{B}\}\right) \Leftrightarrow \left(\mathcal{X}^{\theta_1} \Rightarrow \mathcal{Y}^{\theta_2} \wedge \mathcal{A} \Rightarrow \mathcal{B}\right)$$

Two structural properties may be equivalent even if they appear different because of column equivalences (e.g., via joins) or functional dependencies. Normalization and matching of local properties (sorting and grouping) have been studied extensively in [21,22]. We next review functional dependencies, constraints, and column equivalences, and then describe a number of inference rules to reason with structural properties.

A set of columns $\mathcal{X}$ functionally determines a set of columns $\mathcal{Y}$, if for any two rows that agree on the values of

columns in $\mathcal{X}$, they also agree on values of columns in $\mathcal{Y}$. We denote such *functional dependency* by $\mathcal{X} \rightarrow \mathcal{Y}$.

Functional dependencies appear in several ways:

Trivial FDs: $\mathcal{X} \rightarrow \mathcal{Y}$ whenever $\mathcal{X} \supseteq \mathcal{Y}$.
Key constraints: Keys are a special case of functional dependencies. If $\mathcal{X}$ is a key of relation R, then $\mathcal{X}$ functionally determines every column of R.
Column equality constraints: A selection or join with a predicate $C_1 = C_2$ implies that the functional dependencies $\{C_1\} \rightarrow \{C_2\}$ and $\{C_2\} \rightarrow \{C_1\}$ hold in the result.
Constant constraints: After a selection with a predicate $C = constant$ all rows in the result have the same value for column C. This can be viewed as a functional dependency which we denote by $\emptyset \rightarrow C$.
Grouping columns: After a group-by with columns $\mathcal{X}$, $\mathcal{X}$ is a key of the result and, thus, functionally determines all other columns in the result.

If all tuples of a relation must agree on the values of a set of columns, such columns belong to a *column equivalence class*. An equivalence class may also contain a constant, which implies that all columns in the class have the same constant value. Equivalence classes are generated by equality predicates, typically equijoin conditions and equality comparisons with a constant.

Both functional dependencies and column equivalence classes can be computed bottom up in an expression tree [10, 21,22,26], and in the following, we assume that these have been computed.

Inference rules

We next briefly discuss inference rules for structural properties. The first rule shows that local properties can be truncated.

$$\{*; (A_1, \ldots, A_{m-1}, A_m)\} \Rightarrow \{*; (A_1, \ldots, A_{m-1})\} \tag{1}$$

Global properties cannot be truncated but they can be expanded. A result that is partitioned on columns C_1, C_2 is not partitioned on C_1 because two rows with the same value for C_1 may be in different partitions. However, a result partitioned on C_1 alone is in fact partitioned on C_1, C_2 because two rows that agree on C_1, C_2 also agree on C_1 alone and, consequently, they are in the same partition. More formally,

$$\{\{C_1, \ldots, C_m\}^g; *\} \Rightarrow \{\{C_1, \ldots, C_m, C_{m+1}\}^g; *\} \tag{2}$$
$$\{(C_1^{o_1}, \ldots, C_m^{o_m}); *\} \Rightarrow \{(C_1^{o_1}, \ldots, C_m^{o_m}, C_{m+1}^{o_{m+1}}); *\} \tag{3}$$

If a sequence of rows is sorted, it is also grouped, so:

$$\{*; (A_1, \ldots, C^o, \ldots, A_m)\} \Rightarrow \{*; (A_1, \ldots, \{C\}^g, \ldots, A_m)\} \tag{4}$$
$$\{(C_1^{o_1}, \ldots, C_n^{o_n}); *\} \Rightarrow \{\{C_1, \ldots, C_n\}^g; *\} \tag{5}$$

Functional dependencies allow us to eliminate grouping columns from both global and individual local structural properties:

$$\text{if } \exists\, C \in \mathcal{X} : (\mathcal{X} - \{C\}) \rightarrow C, \text{ then } \mathcal{X}^g \Rightarrow (\mathcal{X} - \{C\})^g \tag{6}$$

The following rule can be applied to eliminate sorting columns in global structural properties:

$$\begin{aligned} &\text{if } \exists\, \{C_1, \ldots, C_{i-1}\} \rightarrow C_i, \text{ then} \\ &\quad \{\{C_1^{o_1}, \ldots, C_{i-1}^{o_{i-1}}, C_i^{o_i}, C_{i+1}^{o_{i+1}}, \ldots\}; \mathcal{A}\} \\ &\qquad \Rightarrow \{\{C_1^{o_1}, \ldots, C_{i-1}^{o_{i-1}}, C_{i+1}^{o_{i+1}}, \ldots\}; \mathcal{A}\} \end{aligned} \tag{7}$$

Finally, the following rule eliminates individual local structural properties altogether:

$$\begin{aligned} &\text{if } \mathcal{X}_1 \cup \ldots \cup \mathcal{X}_{i-1} \rightarrow \mathcal{X}_i, \text{ then} \\ &\quad \{*; (\mathcal{X}_1^{\theta_1}, \ldots, \mathcal{X}_{i-1}^{\theta_{i-1}}, \mathcal{X}_i^{\theta_i}, \mathcal{X}_{i+1}^{\theta_{i+1}}, \ldots)\} \\ &\qquad \Rightarrow \{*; (\mathcal{X}_1^{\theta_1}, \ldots, \mathcal{X}_{i-1}^{\theta_{i-1}}, \mathcal{X}_{i+1}^{\theta_{i+1}}, \ldots)\}. \end{aligned} \tag{8}$$

Property normalization

To simplify property matching, properties are converted to a normalized form. The basic idea of the normalization procedure is as follows. First, in each partitioning, sorting, grouping property, and *functional dependency* replace each column with the representative column in its equivalence class. Then, in each partitioning, sorting and grouping property remove columns that are functionally determined by some other columns. We illustrate this procedure with an example.

Example 4 We want to test whether the structural properties $\mathcal{P}_1 = \{\{C_7, C_1, C_3\}^g; (C_6^{o\uparrow}, C_2^{o\downarrow}, C_5^{o\uparrow})\}$ satisfy the structural properties $\mathcal{P}_2 = \{\{C_1, C_2, C_4\}^g; (\{C_1, C_2\}^g)\}$. We know that the data satisfy the functional dependency $\{C_6, C_2\} \rightarrow \{C_3\}$. There are two column equivalence classes $\{\mathbf{C_1}, C_6\}$ and $\{\mathbf{C_2}, C_7\}$ with C_1 and C_2 as representative columns, respectively. After replacing columns by representative columns, we have that:

$$\mathcal{P}_1 = \{\{C_2, C_1, C_3\}^g; (C_1^{o\uparrow}, C_2^{o\downarrow}, C_5^{o\uparrow})\}$$
$$\mathcal{P}_2 = \{\{C_1, C_2, C_4\}^g; (\{C_1, C_2\}^g)\}$$
$$\{C_1, C_2\} \rightarrow \{C_3\}.$$

Next, we apply the functional dependency to eliminate C_3, which changes $\mathcal{P}_1$ to $\{\{C_2, C_1\}^g; (C_1^{o\uparrow}, C_2^{o\downarrow}, C_5^{o\uparrow})\}$ while $\mathcal{P}_2$ is unchanged.

We first consider global properties. We want to prove that $\{C_2, C_1\}^g \Rightarrow \{C_1, C_2, C_4\}^g$. According to the expansion rule for global properties (inference rule (2)), the implication holds and thus the global properties match. Next, for local properties, we need to show that $(C_1^{o\uparrow}, C_2^{o\downarrow}, C_5^{o\uparrow}) \Rightarrow (\{C_1, C_2\}^g)$. Applying the truncation rule for local properties (inference rule (1)), we obtain $(C_1^{o\uparrow}, C_2^{o\downarrow}) \Rightarrow (\{C_1, C_2\}^g)$ because sorting implies grouping (inference rule (4)). Since both global and local properties match, $\mathcal{P}_1$ satisfy $\mathcal{P}_2$.

5.5 The rule set

A crucial component in our optimizer is the rule set. In fact, the set of rules available to the optimizer is one of the determining factors in the quality of the resulting plans. Many of the traditional optimization rules from database systems are clearly applicable also in our context, for example, removing unnecessary columns, pushing down selection predicates, and pre-aggregating when possible. However, the highly distributed execution environment offers new opportunities and challenges, making it necessary to explicitly consider the effects of large-scale parallelism during optimization. For example, choosing the right partition scheme and deciding when to partition are crucial for finding an optimal plan. It is also important to correctly reason about partitioning, grouping, and sorting properties, and their interaction, to avoid unnecessary computations.

5.5.1 Enforcer rules

Query optimizers in database systems typically start with an optimal serial plan and then add parallelism in a postprocessing step. This approach may result in sub-optimal plans. The challenge is to seamlessly integrate consideration of parallel plans into the normal optimization process. In this section, we describe optimization rules that automatically introduce data exchange operators and thus seamlessly generate and optimize distributed query plans. We enhance the optimization framework in two ways. First, for each logical operator, we consider *both* non-partitioned and partitioned implementations, as long as they can ever satisfy their requirements. Second, we rely on a series of enforcer rules (explained below) to modify requirements for structural properties, say, from non-partitioned to partitioned, or from sorted to non-sorted, etc. Together, with other optimization rules and property inferences, this enables the optimizer to consider both serial and parallel in a *single integrated* framework. It greatly enhances the power of a traditional query optimizer without dramatic infrastructure changes.

We begin with a simple example of sort optimization. Suppose that, during optimization, there is a request to optimize an expression with a specific *local* sort requirement S[2]. The optimizer then considers different alternative physical operators for the root operator of the expression tree, derives what properties their inputs must satisfy, and requests an optimal plan for each input. There are typically three possible ways of ensuring that the result will be sorted. It is up to the optimizer to choose the best plan based on its cost estimates.

- If a physical operator itself can generate a sorted output, try this operator and push requirements imposed by the operator itself to its child expressions.
- If a physical operator retains the input order, try the operator and push the sort requirement plus requirements of the operator itself to its children.
- Otherwise, try the operator but add an explicit sort matching the requirement and then optimize the child expressions *without* the sort requirement.

In the last case, the optimizer *enforces* a sort requirement on top of the physical operator. We call such optimization rules *enforcer* rules. Grouping requirements can be handled in a similar way, except we may not have an explicit group-only operator. A grouped result is usually produced as a side effect of another operator, for example, a one-to-many nested-loop join.

A data exchange operator is similar to sorting. Its only effect is to change structural properties; it does not add or eliminate rows, nor does it modify individual rows in any way. Therefore, we model data exchange operators as *enforcers* of structural properties.

Algorithm 2 shows simplified pseudo-code for enforcing partitioning. For simplicity, handling of sorting and grouping requirements is not shown. When a sorting requirement exists, we consider both sort-merge exchange and regular exchange operations. We also ignore the details of the partitioning requirement.

Although the pseudo-code is much simplified, it captures the core ideas of enforcing partitioning requirements. For any expression with particular partitioning requirements, the optimizer (1) uses an operator that itself satisfies the requirements, (2) uses a partition-preserving operator and pushes the requirements to its children, (3) adds data exchange operators that allow the requirements for its child expressions to be relaxed or modified. The optimizer tries all the alternatives and selects the best plan based on estimated costs.

[2] For global sort requirements, such as those coming from `ORDER BY` clauses, the optimizer considers an alternative that requires its input to be both range partitioned and locally sorted by the original sort columns, which is subsequently handled by the enforcer rules.

Algorithm 2: EnforceDataExchange(*expr*, *reqd*)

```
Input: Expression expr, ReqdProperties reqd
ReqdProperties reqdNew;
Plan plan;
if Serial(reqd) then
    /* Require a serial output                    */
    reqdNew = GenParallel(reqd);
    plan =
    AddExchange_FullMerge(OptimizeExpr(expr,
    reqdNew));
else
    /* Require a parallel output. Enumerate
       all possible partitioning alternatives
       such that X_min ⊆ X ⊆ X_max                */
    cPlans = {};
    foreach valid partition schema X do
        /* Case 1: repartition. Generate new
           partitioning requirements for its
           children; remove specific
           partitioning columns                   */
        reqdNew = GenParallel(reqd);
        cPlans +=
        AddExchange_Repartition(OptimizeExpr(expr,
        reqdNew));
        /* Case 2: initial partition. Force
           the child to generate a serial
           output                                 */
        cPlans += AddExchange_InitialPartition(Opti-
        mizeExpr(expr, reqdNew));

        plan = GetCheapestPlan(cPlans);
    end
end
return plan;
```

Enforcer rules can seamlessly enable a single instance of an operator to work on a partitioned input by introducing a full merge operator, multiple instances of an operator to work on a non-partitioned input by introducing an initial partition operator, and also multiple instances of an operator with specific partitioning requirements to work on any partitioned input sets by introducing a repartition operator. The number of alternatives generated by enforcers could be large so usually the optimizer applies cost-based heuristics to prioritize alternatives and prune out less promising ones.

5.5.2 Other specialized rules

In addition to the rules described earlier, the optimizer has several exploration rules that deal with user-defined operators (e.g., processors and reducers). We next briefly describe some of these specialized rules.

- Pushing selections below processors and reducers. By leveraging pass-through columns (Sect. 4.4.1), we push selection predicates below processors and reducers provided that all predicate columns are annotated as pass-through. This is required even if the input and output schemas of a processor have the same names, as in general the values of such columns in input and output rows might not be correlated.
- Unfolding recursive reducers into local/global pairs. A recursive reducer requires that the input and output schemas be identical. In these situations, we can explore an alternative that performs partial reduction on inputs that are not necessarily partitioned by the reducer columns. These partial results are usually smaller than the raw input, and we can obtain the final result by repartitioning them and performing a final global reducer. This is similar to the traditional local/global aggregate optimization in relational databases.
- Pushing processors and reducers below unions: Similar to the optimization that pushes filters closer to the source tables, the SCOPE optimizer considers alternatives that push down processors below union-all operators. If the operator is a reducer, this optimization cannot always be done, since for correctness the children of the union-all must each have a distinct set of values for the reducing columns. In that case, however, we can leverage the previous optimization and convert a reducer into a local/global pair. The local reducer can be pushed below the union all, effectively pre-aggregating data and resulting in a more efficient plan. Also, if the optimizer can deduce that the values of the reducing columns on each children of a union-all operator are disjoint (leveraging constraints or script annotations), the optimizer can effectively push the reducer below the union all without requiring a global reducer afterwards, greatly improving overall performance of the resulting execution plans.

6 Code generation and runtime engine

The SCOPE runtime engine implements a rich class of composable physical operators. It supports various implementation of relational operations, such as filtering, projection, sorting, aggregation, and join. For instance, SCOPE provides three join operators: sort-merge join, (index) nested-loop join, and hash join. The engine contains a few built-in and optimized UDOs for common tasks. For example, *DefaultTextExtractor* and *DefaultTextOutputter* are provided for users to extract inputs from and output results to text files. The engine supports different access methods for structured streams, including table scan and index lookup.

Each SCOPE operator implements *open*, *next*, and *close* functions that allow a parent operator to get the result *one* tuple at a time. Such database-like iterator interface supports pipelining of operators naturally.

After optimization, a physical execution tree is generated. The runtime engine performs the following tasks. First, the

engine walks through the plan bottom up and *generates* code for the corresponding operator, based on internal operator template. Then, it combines a series of physical operators into a *super vertex*, obtained by splitting the output tree at exchange and spool operators. In addition to these mandatory super vertex boundaries, it sometimes breaks large super vertices into smaller ones to avoid having execution units that run for a long time (and thus would make recovery in the presence of failures more time-consuming). All the generated code is compiled into an assembly, and the entry function for each super vertex is written into a super vertex definition file. Each super vertex is scheduled and executed in one machine. Operators within a super vertex are executed in a way very similar to a single-node database engine. Specifically, every super vertex is given enough memory to satisfy its requirements (e.g., hash tables or work tables for sorts), up to a certain fraction of total available memory and a fraction of the available processors. This procedure sometimes prevents a new super vertex from being run immediately on a busy machine. Similar to traditional database systems, each machine uses admission control techniques and queues outstanding super vertices until the required resources are available. As we discuss later, in contrast to DBMSs, the overall system has additional scheduling choices for such queued super vertices that reduce overall latency.

The output of the compilation of a script thus consists of three components: (1) the algebra file that enumerates all super vertices and the data flow relationships among them, (2) the super vertex definition file, which contains the specification and entry points in the generated code for all super vertices, and (3) the assembly itself, which contains the generated code. This package is sent to the cluster for execution.

The SCOPE code generation mechanism is flexible and can generate specific and highly optimized code for a script. The system knows precisely the input and output schema for each operator. The generated code is specific to the schemas and completely avoids any runtime tuple interpretation. The system also has many opportunities to further optimize the generated code. For instance, the intermediate record format can be changed to address the needs of subsequent operators.

6.1 Execution model

During execution, a super vertex reads inputs either locally or remotely. Operators within a super vertex are executed in a pipelined fashion. The final result of a super vertex is written to local disks, waiting for the next vertex to *pull* data. This *pull* execution model is much like to the MapReduce execution model. However, the SCOPE engine does not require every function to be mapped into a super vertex as in the MapReduce model, potentially in a unnatural way. The optimizer has great flexibility to break arbitrary functions into super vertices based on a principled cost model.

This execution model is very different from that of traditional parallel databases, which is optimized to avoid hitting disks and instead transfer intermediate data on the fly. Passing data on the fly requires that both producer and consumer machines are running concurrently and extensive synchronization between them. This approach does not scale to thousands of machines and is less reliable when one of the involved machines can crash in the middle of execution.

The *pull* execution model and materializing intermediate results to disk may sound inefficient at first glance. However, it has 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. As we shall describe in Sect. 7, the job manager guarantees the two vertices are executed in sequence and has great flexibility when to schedule the vertices. Second, all intermediate results are written to local disks. In case of vertex failures, which are inevitable, all that is required is rerun the failed vertex from the cached inputs on disks. Only a small portion of a query plan may need to be re-executed. Finally, writing intermediate results to disks frees system memory to execute other vertices and simplifies computation resource management.

6.2 Runtime data exchange

We next describe in some detail the support of data exchange operations in the SCOPE runtime engine. Having efficient implementations of such operators is crucial as data shuffling is a common and expensive operation. The SCOPE runtime engine supports multiple variants of partitioning and merging operators. For merging operators, the system attempts to overlap network transfer with subsequent operations as much as possible.

For partitioning operators, SCOPE supports hash and range partitioning over one or more columns. The total number of partitions can be either predefined or determined at runtime based on intermediate data size.

Hash partitioning: A hash value is computed from the partitioning columns and taken modulo the number of total partitions to get the destination partition.

Range partitioning: Calculating partitioning boundaries that produce partitions of uniform size is crucial for avoiding outliers and achieving high query performance but challenging in a distributed fashion. Figure 7 presents a job graph that the SCOPE uses to range partition the input into 250 buckets. First, each input machine generates a local histogram over its local data. Second, a single coordinator combines all the local histograms and calculate the partition boundaries based on the global histogram (a detailed description of histogram computation is beyond the scope of this paper). Finally, partitioning information is broadcast to all the input

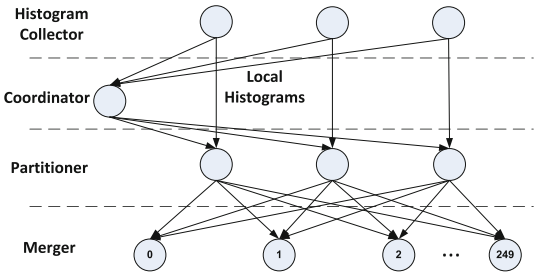

Fig. 7 Range partitioning

machines, which then partition their local data accordingly. This dynamic partitioning strategy greatly improves partitioning quality.

Index-based partitioning: Independent of partitioning type, the partitioned outputs are first written to local disks in separate files. When the number of partitions is large, writing too many intermediate files at the same time incurs lots of random I/O and significantly increases the overall latency. The SCOPE runtime engine supports a novel partitioning operation that relies on the indexing provided by structured streams. For each tuple, the destination partition ID, called "PID," is calculated using the partition function in a normal way and is added to the tuple as a virtual column. A stable sort on PID is performed, so that the original tuple order within the same PID value is maintained. (This is crucial when maintaining the tuple order within each partition is required for subsequent operators). The sorted results are finally bulk written into a single index file. The following merging operator reads the corresponding partition by index lookup on the desired PID. Index-based partitioning allows the system to partition large data into thousands of partitions without penalizing the I/O system.

7 Job scheduling

Execution of a SCOPE job is coordinated by a job manager (JM), adapted from Dryad [17]. The JM is responsible for constructing the job graph and scheduling work across the available resources in the cluster. As described in Sect. 6, a SCOPE execution plan consists of a DAG of SCOPE super vertices that can be scheduled and executed on different machines independent of each other. The JM groups distinct types of vertices into separate *stages* to simplify job management. All the vertices in a stage perform the same computation, defined in the query plan, on a different partition of input data. The JM maintains the job graph and keeps track of the state and history of each vertex in the graph.

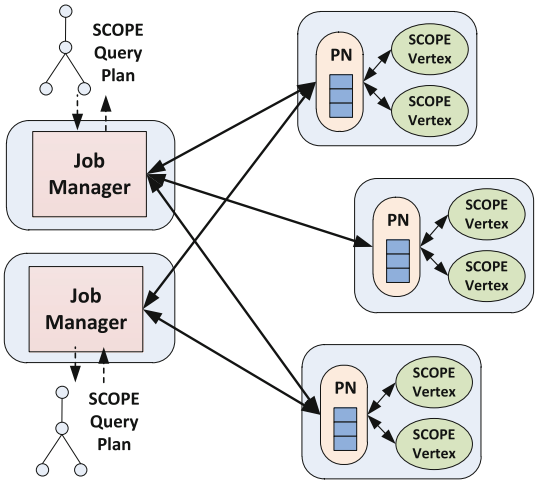

Fig. 8 SCOPE job scheduling

7.1 Job manager overview

Figure 8 shows a simplified overview of scheduling a SCOPE job. Each machine in the cluster is capable of both data storage and computation. Besides storage services (not shown in the figure), a processing node (PN) service runs on each machine to manage local system resources (e.g., memory and disk) and schedules execution of local vertices. PN nodes maintain a local queue of vertex execution requests from different JMs and create processes on behalf of the JMs taking into account vertex priorities and available resources. The first time a vertex is executed on a machine, its assembly is sent from the JM to the PN and subsequently it is executed from a cache. PN also monitors the state of each running vertex, such as whether it succeeds or fails and how much data have been read and written, and communicates the information back to the JM periodically through a heartbeat mechanism. This feedback mechanism allows the JM to monitor the time a super vertex is queued a busy PN machine; If necessary, the JM can schedule duplicate instances of long-delayed vertices on different machines to reduce latencies. Whenever the first instance of a super vertex starts executing, all duplicate instances are eliminated from the corresponding PN queues.

When the job starts, the system picks one machine to launch the JM, which becomes responsible for scheduling the SCOPE job. An initial DAG is built to track relationship among different vertices. As we shall describe later in Sect. 7.2, the job graph may adapt itself dynamically as the execution proceeds and system environment changes. When all of a vertex's inputs 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 scale back to another machine that is close in network topology (possibly in the same rack), so reading the input can be done efficiently with minimum network traffic. A SCOPE vertex may also specify a "constraint" listing a preferred set of machines to run. Once the machine is chosen, the JM places a request to run the vertex, together with its assemblies, onto its PN queue.

For each vertex, the JM keeps track of all of inputs' availability and locations. It has no information about the actual computation performed by the vertex but it receives initial estimates of resource consumption from the optimizer. As the job proceeds, the JM tracks actual resource consumption for vertices in a stage. This information is used to refine resource estimates for the remaining vertices in the same stage to improve the local scheduling decisions made by PNs.

When a vertex execution fails for any reason, the JM is informed. The failure could be due to user code errors or system transient failures, such as machine reboot or network temporal congestion. Depending on the failure type, the JM applies different policies and acts accordingly, including failing the job or retrying the failed vertex. A failed vertex can be rerun without rerunning the entire job [17].

There are several other interesting aspects of the JM. Multiple instances of vertex may be scheduled speculatively, anticipating system transient failures. It is important for the JM to detect and act on vertex outliers early in order to avoid prolonged job runtime [2]. Hundreds of SCOPE jobs are often running concurrently, so scheduling them with fine-grain resource sharing is done by a global scheduler [18].

7.2 Runtime optimizations

Accurate information such as data location or system environment properties are not always available at compilation time. Similar to traditional databases, the SCOPE optimizer generates the optimal plan "skeleton" based on query semantics but certain decisions are better left to runtime when additional information is available. In this section, we briefly describe two dynamic optimizations to illustrate the interactions between the SCOPE runtime engine and the JM at query runtime.

A large cluster typically has a hierarchically structured network. Each rack of machines has its own dedicated mini-switch, and the per-rack switches are then connected to a single common switch. In this network topology, it is important to not overload the common switch and use the per-rack switches as much as possible. Whenever possible, the JM schedules vertices to execute on the same machine as their input data or at least within the same rack as the data.

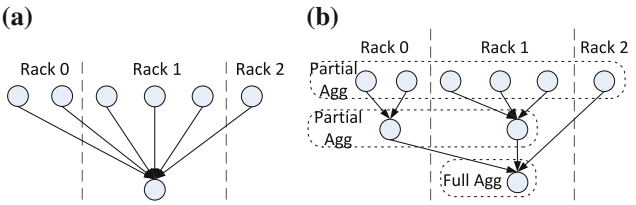

Fig. 9 Dynamic aggregation optimization. **a** Before optimization, **b** after optimization

Dynamic aggregation optimization As described in Sect. 5, the last stage of data exchange operation requires aggregating inputs from a large number of source machines. The number of source machines could be hundreds or even thousands for a large data set. This scale presents challenges on how to efficiently aggregate data distributed among machines connected by a hierarchal network.

Figure 9 shows a simplified example of aggregating inputs from 6 machines in 3 racks. The default approach, shown in Fig. 9a, is to aggregate all the inputs together in a single vertex. The approach works fine when the number of inputs is relatively small but not when there are thousands of inputs. Reading from thousands of remote machines concurrently across the network could result in a network traffic spike and suffer from high probability of network failures. Furthermore, the merge vertex could be quite expensive to execute, so in case of failure, the cost of rerunning the vertex would be high and the likelihood of encountering another network failure would be high.

Figure 9b shows the job graph as refined at runtime. The JM includes heuristics to add intermediate aggregation vertices at runtime so as to ensure that each vertex has no more than a set number of inputs or a set volume of input data to avoid overloading the I/O system of the vertex. This works well for some SCOPE queries with merge operation followed by partial aggregation. As partial aggregation reduces the input data size and can be applied multiple times at different levels without changing the correctness of the query, it makes sense to aggregate the inputs within the same rack before sending them out, thereby reducing the overall network traffic among racks.

Dynamic broadcast optimization When joining a large input with small input, the SCOPE optimizer may choose a broadcast join. Typically, the large input is distributed among several machines while the small input is located in a single machine. Figure 10 illustrates an example of broadcasting the small input from one machine to 5 different machines across the cluster.

The default approach, shown in Fig. 10a, is to let the JM choose the vertex placement based on data locality. It could end up placing all 5 join vertices into rack 2 and overload the system or transferring the data from $input2$ five times

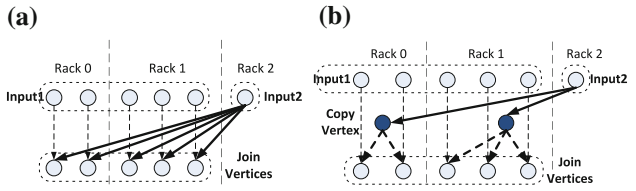

Fig. 10 Dynamic broadcast optimization. **a** Before optimization, **b** after optimization

(shown in solid arrows) across racks, wasting the scarce network bandwidth. With dynamic broadcast optimization, shown in Fig. 10b, the JM can detect that multiple vertices reside in one destination rack. A "copy" vertex is created for the rack that copies the data across racks *once*. Any vertices in the destination rack can now read the data cheaply from the copy vertex (shown in dashed arrows) within the same rack. In this case, the data are only transferred across racks twice with two copy vertices, saving network bandwidth. Even with overhead of copy vertices, the saving typically translates into overall improvements in query latency. This optimization cannot be done at compilation time as the data locations are unknown until runtime. However, SCOPE is able to identify the pattern and explicitly include a dynamic broadcast manager at runtime.

8 Case studies

At Microsoft, tens of thousands of SCOPE jobs are executed daily that read and write tens of petabytes of data powering the whole online services division. The number of daily jobs has doubled every six months for the past two years, putting the system performance and scalability to test every day. Due to business and IP reasons, we cannot include absolute performance numbers that could reveal confidential business intelligence unless they have been publicly disclosed elsewhere. Thus, in this section, we include one case study showing the benefits of query optimization. In addition, we report performance results on a known benchmark. Some additional performance results can be found in [7].

8.1 Benefits of systematic query optimization

We show an example of a SCOPE script that is used internally for system diagnosis. During execution of all SCOPE jobs, the system logs various types of *events*. Two examples of those events that we use in our example are `VertexStarted` and `VertexEnded`. As their names suggest, `VertexStarted` logs information when a vertex process starts to run. The information includes the machine name on which the vertex runs, the job GUID, the vertex GUID, vertex priority, and the time stamp when the vertex starts. Similarly, `VertexEnded` logs information when a vertex process terminates. Due to the distributed environment, raw events may be duplicated and thus log processing must perform duplicate elimination. The script below calculates how much machine time has been spent on jobs issued by different user groups during last month. This is a rough estimate of the system resources used by each user group.

```
startData =
  SELECT DISTINCT CurrentTimeStamp AS
         StartTime, VertexGUID
  FROM "VertexStartEvents?Date=(Today-30)...
         Today"
  USING EventExtractor("VertexStarted");

endData =
  SELECT DISTINCT CurrentTimeStamp AS EndTime,
         UserGroupName, VertexGUID
  FROM "VertexEndEvents?Date=(Today-30)...
         Today"
  USING EventExtractor("VertexEnded");

  SELECT UserGroupName,
         SUM((EndTime-StartTime).TotalHours) AS
         TotalCPUHours
  FROM startData JOIN endData
     ON startData.VertexGUID == endData.Vertex
       GUID
  GROUP BY UserGroupName
```

The script first selects events from last month and extracts time stamp information when each vertex starts and ends, respectively, plus the vertex GUID and user group information. Next, duplicates in the raw events are removed by applying a `DISTINCT` aggregate on *all* columns. Finally, the cleansed data are joined on the vertex GUID and the total CPU time per user group is calculated.

Figure 11 compares query plans with and without optimization. A partitioned stream is shown as three arrows. Partitioning operators are shown with a dark gray background. A partitioning operator may consist of two sub-operators—the first generating partitions on source machines and the second merging corresponding source partitions on destination machines. A sequence of operators between two partitioning operators are grouped into a super vertex, which is depicted in light gray circles with annotated SV numbers.

The default plan in Fig. 11a performs the following steps. It extracts the `VertexStarted` event, sorts this input on each machine by the grouping columns $\{starttime, guid\}$ and removes duplicates locally (local streaming aggregation) to reduce data before hitting the network in the next stage. Similar operations are performed on columns $\{endtime, usergroup, guid\}$ for the `VertexEnded` stream. Then, both the intermediate results are repartitioned by the grouping columns, so all rows with the same value of the grouping columns end up on the same destination machine. Each destination machine sort-merges its inputs from the source machines, so the sort order is maintained.

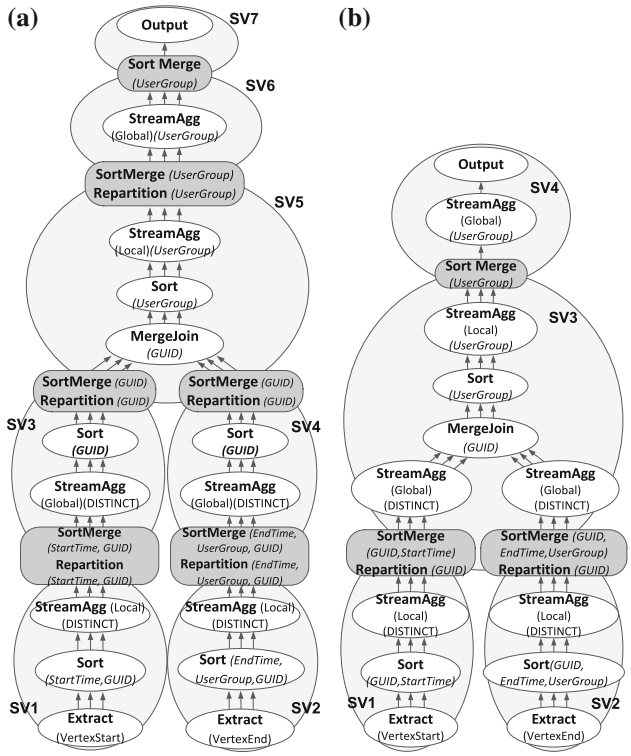

Fig. 11 Query plan comparison. **a** Default plan, **b** optimized plan

Similar operations apply to both inputs, except that one side is partitioned and sort-merged by $\{starttime, guid\}$ and the other side is by $\{endtime, usergroup, guid\}$. After the repartitioning, the global aggregates are then calculated in parallel. On each machine, the results are sorted again by the $guid$ column in preparation for the subsequent merge join. The intermediate results are repartitioned by the join column $guid$. Each destination machine sort-merges its partitions to maintain the sort order on $guid$ and the results flow directly into the merge join instance on that machine. At this point, join results are resorted locally on the subsequent grouping column $usergroup$ and a local aggregate is applied immediately to reduce the data. The data are subsequently repartitioned on $usergroup$, and each destination machine sort-merges its inputs to retain the sort order on $usergroup$. All rows from the $usergroup$ now reside on the same machine, and the global aggregate on $usergroup$ is calculated. Finally, all results are aggregated to a single machine to produce the final results.

Figure 11b shows the plan generated by the optimizer. We first determine input properties required by individual operators. Based on the rules in Table 6, the partitioned merge join requires both inputs to have structural properties $\{\{guid\}^{\theta}; (guid^{o})\}$. The partitioned stream aggregate on `VertexStarted` events requires the input to satisfy $\{\mathcal{X}^{\theta}; (\{starttime, guid\}^{g})\}$, where $\emptyset \subset \mathcal{X} \subseteq \{starttime, guid\}$ and the partitioned stream aggregate on `VertexEnded` events requires the inputs to satisfy $\{\mathcal{Y}^{\theta}; (\{endtime, usergroup, guid\}^{g})\}$, where $\emptyset \subset \mathcal{Y} \subseteq \{endtime, usergroup, guid\}$.

In Fig. 11b, inputs are extracted and sorted by columns $(guid^{o}, starttime^{o})$ and $(guid^{o}, endtime^{o}, usergroup^{o})$, respectively, and then hash repartitioned on $guid$. Based on the rules in Table 5, the results have properties $\{\{guid\}^{g}; (guid^{o}, starttime^{o})\}$ and $\{\{guid\}^{g}; (guid^{o}, endtime^{o}, usergroup^{o})\}$, respectively, which satisfy the requirements of its `DISTINCT` aggregate. The outputs from the `DISTINCT` aggregates, in turn, also satisfy $\{\{guid\}^{g}; (guid^{o}, starttime^{o})\}$ and $\{\{guid\}^{g}; (guid^{o}, endtime^{o}, usergroup^{o})\}$, respectively. The properties satisfy the requirements of the partitioned merge join, so there is no need to resort or repartition the inputs before joining. The join results are small, so the optimizer decides not to repartition on $usergroup$. Instead, it sort-merges the inputs into a single serial output and performs the global aggregate on a single machine.

Compared with the default query plan, the optimized plan runs 1.9× faster. This result clearly shows the benefits of a high-level declarative language coupled with a cost-based optimizer. It would be difficult and time-consuming to manually tune individual MapReduce tasks, especially when the pipeline they are part of needs to evolve to satisfy new requirements.

8.2 Performance micro-benchmark

To showcase raw performance numbers, we next report results on sorting one terabyte of uniformly distributed data, composed of 100-byte rows (20-byte key and 80-byte payload). The input data are distributed in a test cluster of 1,000 machines. Each machine has two six-core AMD Opteron processors running at 1.8 GHz, 24 GB of DRAM, and four 1 TB SATA disks. All machines run Windows Server 2008 R2 Enterprise X64 Edition. The SCOPE script is as follows:

```
D = EXTRACT c1, c2 FROM "input.txt"
    USING DefaultTextExtractor();

S = SELECT * FROM D ORDER BY c1;

OUTPUT TO "output.txt"
    USING DefaultTextOutputter();
```

Figure 12 shows the execution plan for the script above and some execution statistics. SV1 vertices read and partition the input in parallel. Based on the uniformity distribution assumption, the range repartition operator calculates partition boundaries at compile time. SV2 vertices gather the corresponding buckets across the network and perform a local sorting before writing the final result back to disk. Local sorting tries to overlap with reads to hide network latency. Overall, the latency for the job is 57 s, which roughly corresponds to sorting over 17GB per second.

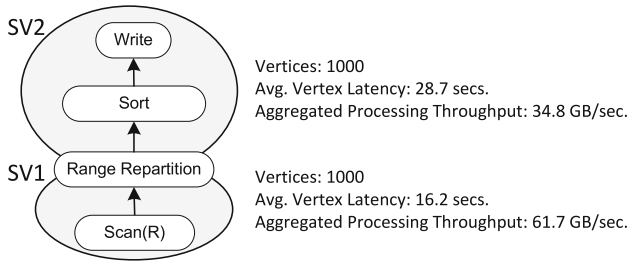

Fig. 12 Sorting 1 TB of data in scope

9 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 [13], Hadoop Distributed File System [3], and Microsoft Cosmos [7] provide scalable and fault-tolerant storage solutions. Simple programming models have also been designed for distributed computing. For instance, MapReduce [11] 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 [17] provides additional flexibility where a distributed computation job is represented as a dataflow graph and supports scripting languages that allow users to easily compose distributed data-flow programs. These programming models help programmers write distributed applications, but still require developers to deal with implementation details like data distribution and system configuration to achieve good performance. A MapReduce program written in C++ is longer than the corresponding SCOPE script (e.g., a word count example [11] requires 70 lines of C++ and 6 lines of SCOPE).

To tackle these issues, higher level programming languages plus conceptual data models have been proposed. Pig [23] is a system developed by Yahoo! and Apache to support large-scale data processing. Users write applications in a data flow language using a nested data model. Each step in a program specifies a single, high-level data transformation. A complex computation is expressed as a series of such transformations. A Pig program is compiled by the underlying system into a sequence of MapReduce operators that are executed using Hadoop. Hive [28,29] is another open-source solution built on top of Hadoop, which supports a SQL variant. Programs written in Hive are transformed into a set of MapReduce jobs by a rule-based query optimizer. In addition to query operator like select, project, join, aggregation, Hive supports both data definition and data manipulation languages. Both Pig and Hive use a MapReduce execution environment, which requires every computation to be structured as a sequence of map-reduce pairs. In SCOPE , any computation can be expressed as an acyclic dataflow graph and each node can execute one or a pipeline of operations, which results in a richer execution model.

Other high-level languages/systems were recently proposed in the community. Sawzall [24] is a procedural language mimicking database *cursors*, which traverses through a rowset and processes one row at a time. Jaql [5] is a declarative scripting language on top of MapReduce, which uses a JSON-like data model. Dremel [20] is designed for interactive ad-hoc aggregation queries on read-only nested columnar data. Tenzing [8] is a SQL-based query engine built on top of MapReduce for ad-hoc data analysis. DryadLINQ [31] integrates Dryad with .NET Language INtegrated Query (LINQ). The Nephele/PACTs system [4] extends the MapReduce programming model by adding second-order operators that *declare* parallelization contracts. Finally, the Hyracks system [6] improves the MapReduce runtime system by representing a job as a DAG of operators and connectors.

HadoopDB [1] uses Hadoop as the communication layer to connect shared-nothing processing nodes running instances of PostgreSQL. It performs query compilation and processing in two stages. Queries expressed in Hive are transformed into MapReduce programs by Hive's query compiler. Before execution, the *SMS planner* intercepts the execution plan and finds the largest subtrees that can be pushed to individual PostgreSQL servers without repartitioning the data. The results returned by PostgreSQL may still need to be aggregated or further processed using Hive.

The SCOPE system has a powerful cost-based query optimizer that considers and optimizes 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. The SCOPE system 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.

Many of the core optimization techniques that the SCOPE optimizer implemented originated from early research projects in parallel databases [12] and traditional databases [16,19,21,22,25,26,30]. The SCOPE optimizer enhances previous work in several ways. For example, it fully integrates parallel plan optimization and reasoning with more complex properties derivation relying on structural data properties.

10 Conclusion

In this paper, we present SCOPE, a scripting language for massive data analysis on large clusters. The SCOPE language is high-level and declarative, with a strong resemblance to SQL, so as to provide a single machine programming abstraction

Table 7 SCOPE innovations inspired from both parallel databases and MapReduce systems

	Parallel databases	MapReduce/dryad
Data representation	Schemas	Extractors and outputters
	Structured streams (tables)	
	Partitioning, indexing, affinity, column groups	
Programming model	Relational algebra	MapReduce extensions
	Declarative language	.NET integration, User defined types
Compilation and optimization	Query optimization	Code generation
	Structural properties, parallel plan generation, etc.	Super vertex generation
Runtime and optimization	Intra-vertex pipelining/iterator model	Inter-vertex pull model
	Relational physical operators	Job scheduling (Fault tolerance, outlier detection)
	Index-based execution strategies	Runtime graph optimizations

and enable systematic and transparent optimization. Rich structural properties and access methods allow sophisticated query optimization and efficient query processing. At the same time, SCOPE is also highly extensible. Users can easily create customized processors, reducers, and combiners by extending corresponding built-in C# interfaces. Such extensions allow users to efficiently solve problems that are otherwise difficult to express in SQL. SCOPE incorporates the best characteristics from parallel databases and MapReduce systems, as summarized in Table 7, achieving both good performance and scalability. The SCOPE system is running in production over tens of thousands of machines at Microsoft, serving a variety of Microsoft online services.

Acknowledgments We would like thank the following people for their contributions to the SCOPE system: Roni Burd, Jon Fowler, Pat Helland, Sapna Jain, Bob Jenkins, Pavel Iakovenko, Wei Lin, Jingwei Lu, Martin Neupauer, Thomas Phan, Bill Ramsey, Bikas Saha, Bing Shi, Liudmila Sudanava, Simon Weaver, Reid Wilkes, Fei Xu, Eugene Zabokritski, and Grace Zhang. We would also like to thank all the members of Bing infrastructure team for their support and collaboration.

References

1. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., Rasin, A.: HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In: Proceeding of VLDB Conference (2009)
2. Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., Saha, B., Harris, E.: Reining in the outliers in map-reduce clusters using Mantri. In: Proceedings of OSDI Conference (2010)
3. Apache. Hadoop. http://hadoop.apache.org/
4. Battré, D., Ewen, S., Hueske, F., Kao, O., Markl, V., Warneke, D.: Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In: Proceedings of the ACM Symposium on Cloud Computing (2010)
5. Beyer, K.S., Ercegovac, V., Gemulla, R., Balmin, A., Eltabakh, M., Kanne, C.-C., Ozcan, F., Shekita, E.J.: Jaql: a scripting language for large scale semistructured data analysis. In: Proceedings of VLDB Conference (2011)
6. Borkar, V., Carey, M., Grover, R., Onose, N., Vernica, R.: Hyracks: a flexible and extensible foundation for data-intensive computing. In: Proceedings of ICDE Conference (2011)
7. Chaiken, R., Jenkins, B., Larson, P.-Å., Ramsey, B., Shakib, D., Weaver, S., Zhou, J.: SCOPE: easy and efficient parallel processing of massive data sets. In: Proceedings of VLDB Conference (2008)
8. Chattopadhyay, B., Lin, L., Liu, W., Mittal, S., Aragonda, P., Lychagina, V., Kwon, Y., Wong, M.: Tenzing: a SQL implementation on the MapReduce framework. In: Proceedings of VLDB Conference (2011)
9. Copeland, G.P., Khoshafian, S.N.: A decomposition storage model. In: Proceedings of SIGMOD Conference (1985)
10. Darwen, H., Date, C.: The role of functional dependencies in query decomposition. In: Relational Database Writings 1989-1991. Addison Wesley (1992)
11. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proceedings of OSDI Conference (2004)
12. DeWitt, D., Gray, J.: Parallel database systems: the future of high performance database processing. Commun. ACM **35**(6) 85–98 (1992)
13. Ghemawat, S., Gobioff, H., Leung, S.-T.: The Google file system. In: Proceedings of SOSP Conference (2003)
14. Graefe, G.: Encapsulation of parallelism in the Volcano query processing system. In: Proceeding of SIGMOD Conference (1990)
15. Graefe, G.: The Cascades framework for query optimization. Data Eng. Bull. **18**(3) 19–29 (1995)
16. Graefe, G., McKenna, W.J.: The Volcano optimizer generator: extensibility and efficient search. In: Proceeding of ICDE Conference (1993)
17. Isard, M. et al.: Dryad: distributed data-parallel programs from sequential building blocks. In: Proceedings of EuroSys Conference (2007)
18. Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., Goldberg, A.: Quincy: fair scheduling for distributed computing clusters. In: Proceedings of SOSP Conference (2009)
19. Lu, H., Ooi, B.-C., Tan, K.L.: Query Processing in Parallel Relational Database Systems. IEEE Computer Society Press, Los Alamitos (1994)
20. Melnik, S., Gubarev, A., Long, J.J., Romer, G., Shivakumar, S., Tolton, M., Vassilakis, T.: Dremel: interactive analysis of webscale datasets. In: Proceedings of VLDB Conference (2010)
21. Neumann, T., Moerkotte, G.: A combined framework for grouping and order optimization. In: Proceedings of VLDB Conference (2004)

22. Neumann, T., Moerkotte, G.: An efficient framework for order optimization. In: Proceedings of ICDE Conference (2004)
23. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: Proceedings of SIGMOD Conference (2008)
24. Pike, R., Dorward, S., Griesemer, R., Quinlan, S.: Interpreting the data: parallel analysis with sawzall. Sci. Program. J. **13**(4) 277–298 (2005)
25. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: Proceedings of SIGMOD Conference (1979)
26. Simmen, D., Shekita, E., Malkenus, T.: Fundamental techniques for order optimization. In: Proceedings of SIGMOD Conference (1996)
27. Stonebraker, M., Abadi, D., DeWitt, D.J., Madden, S., Paulson, E., Pavlo, A., Rasin, A.: MapReduce and parallel DBMSs: friends or foes? Commun. ACM **53**(1) 64–71 (2010)
28. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive—a warehousing solution over a MapReduce framework. In: Proceedings of VLDB Conference (2009)
29. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Antony, S., Liu, H., Murthy, R.: Hive—a petabyte scale data warehouse using Hadoop. In: Proceedings of ICDE Conference (2010)
30. Wang, X., Cherniack, M.: Avoiding sorting and grouping in processing queries. In: Proceeding of VLDB Conference (2003)
31. Yu, Y. et al.: DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In: Proceedings of OSDI Conference (2008)
32. Zhou, J., Larson, P.-Å., Chaiken, R.: Incorporating partitioning and parallel plans into the SCOPE optimizer. In: Proceedings of ICDE Conference (2010)
33. Zhou, J., Larson, P.-Å., Freytag, J.-C., Lehner, W.: Efficient exploitation of similar subexpressions for query processing. In: Proceedings of SIGMOD Conference (2007)