BIG DATA MINING AND ANALYTICS
ISSN 2096-0654 04/06 pp252–265
Volume 4, Number 4, December 2021
DOI: 10.26599/BDMA.2021.9020009

LotusSQL: SQL Engine for High-Performance Big Data Systems

Xiaohan Li, Bowen Yu, Guanyu Feng, Haojie Wang, and Wenguang Chen*

Abstract: In recent years, Apache Spark has become the de facto standard for big data processing. SparkSQL is a module offering support for relational analysis on Spark with Structured Query Language (SQL). SparkSQL provides convenient data processing interfaces. Despite its efficient optimizer, SparkSQL still suffers from the inefficiency of Spark resulting from Java virtual machine and the unnecessary data serialization and deserialization. Adopting native languages such as C++ could help to avoid such bottlenecks. Benefiting from a bare-metal runtime environment and template usage, systems with C++ interfaces usually achieve superior performance. However, the complexity of native languages also increases the required programming and debugging efforts. In this work, we present LotusSQL, an engine to provide SQL support for dataset abstraction on a native backend Lotus. We employ a convenient SQL processing framework to deal with frontend jobs. Advanced query optimization technologies are added to improve the quality of execution plans. Above the storage design and user interface of the compute engine, LotusSQL implements a set of structured dataset operations with high efficiency and integrates them with the frontend. Evaluation results show that LotusSQL achieves a speedup of up to $9\times$ in certain queries and outperforms Spark SQL in a standard query benchmark by more than $2\times$ on average.

Key words: big data; C++; Structured Query Language (SQL); query optimization

1 Introduction

The rapid development of information technology has brought significant progress to human society, and the amount of data that computer systems need to deal with has increased accordingly. Different frameworks[1–6] have been proposed for big data processing, and they include Apache Spark[4], which has become the de facto standard in recent years. Spark's core programming abstraction is in the form of an immutable object collection called Resilient Distributed Dataset (RDD). Although the Spark RDD API offers powerful semantics that can support complex data analytics, it is not easy enough to use and often requires manual optimization.

Structured Query Language (SQL) is a common choice for data analysis in many scenarios. It is originally designed for relational databases, but many big data systems adopt it as a powerful tool for Online Analysis Processing (OLAP) applications. SQL offers a straightforward interface and gives the potential to optimize the original workload and deliver superior execution performance. To evaluate the execution efficiency of SQL queries, the Transaction Processing Performance Council (TPC) defines a set of benchmarks, among which TPC-H is widely used for OLAP performance evaluation.

SparkSQL[7] is designed for processing structured data on Spark. It provides a bridge between relational tables and RDDs, and it can function like a distributed SQL query engine, thus bringing significant convenience for end-users. SparkSQL also leverages careful optimization to expedite queries. However, SparkSQL suffers from the inefficiency of Spark. Many analyses[8–11] indicate the shortcomings of Spark. Some overheads, including garbage collection and data serialization, are attributable to Java Virtual Machines (JVMs) while others are from

• Xiaohan Li, Bowen Yu, Guanyu Feng, Haojie Wang, and Wenguang Chen are with the Department of Computer Science and Technology, Tsinghua University, China. E-mails: {xh-li18, yubw15, fgy18, wang-hj18}@mails.tsinghua.edu.cn, cwg@tsinghua.edu.cn.

* To whom correspondence should be addressed.

Manuscript received: 2021-05-11; accepted: 2021-05-28

Spark's design, including its element-wise function calls.

Flare[9], an accelerator module for Spark whose performance is at par with main-memory database HyPer, delivers order-of-magnitude speedups to SparkSQL on single-core TPC-H computing. The distributed relational database HRDBMS[12] is several times faster than SparkSQL in terms of the total elapsed time of TPC-H.

To bypass the inefficiency of JVM and Spark, scholars have proposed different big data frameworks[5, 13] that are based on native languages. Lotus is a high-performance data-parallel computing engine built with C++. Lotus gains high performance because of its bare-metal runtime environment, compact storage strategy, coarse-grained function call, and memory-efficient design. Details about Lotus can be found in Section 2.1. Although C++ brings significant benefits for Lotus, it also entails extensive programming and debugging efforts. Features in modern C++, such as template usage and automatic type deduction are widely-used in Lotus. Although APIs are carefully designed, users might encounter confusing problems when constructing complex applications. Thus, a high-level interface such as SQL is desired.

Supporting SQL with Lotus involves two main challenges. First, a semantic gap exists between Lotus and SQL. Lotus's primary dataset abstraction organizes data as one-dimensional, whereas a table in SQL is two-dimensional. Second, building SQL engines with full functionality that delivers optimized execution performance requires massive development efforts.

In this work, We present LotusSQL, an SQL module for Lotus, which addresses the aforementioned challenges. For the first challenge, we design and implement a set of structured data operators on Lotus dataset abstraction to meet the need for table manipulations. Column storage and operation fusion are used to improve performance. For the second one, we integrate the open-source framework Calcite[14] as a frontend, which provides the parser, the validator, and basic optimizer architectures to deal with SQL queries. LotusSQL adopts a cost-based optimizer of Calcite and offers cost evaluation methods for Lotus operators to guide the optimization. In addition, LotusSQL extends Calcite's optimization process to produce good execution plans.

The main contributions of this work are as follows:

- A set of efficient structured data operations are designed and implemented upon a native dataset backend.
- Calcite is extended with customized cost models and optimization passes to obtain a full-featured SQL parser and optimizer for Lotus at reasonable development efforts.
- Performance experiments on the complete set of TPC-H queries are conducted, and related analyses are performed to evaluate LotusSQL.

The rest of the paper is organized as follows. Section 2 introduces the background knowledge of two important systems related to our work. Section 3 presents an overview of the workflow of LotusSQL. Section 4 explains the design of physical operators and the cost model. Section 5 formulates additional query optimization techniques used in our SQL engine. Then, Section 6 details the performance of LotusSQL on real workloads. Section 7 discusses our design choice in two aspects. Section 8 introduces related works. Finally, Section 9 concludes the study and identifies future research directions.

2 Background

2.1 Lotus

Lotus is a single-machine data parallel computing engine constituted by a low-overhead storage module and a highly efficient compute module. The storage module is designed to have a low overhead on the basis of a combination of buffer caches and compact object models. The compute module is a C++ dataset programming model. Lotus datasets provide the abstraction of compact collections and efficient operation implementations.

Lotus dataset is logically an array of records that is segmented into multiple partitions. Except for distributed allocation, its abstraction is quite similar to Spark's RDD. It also adopts a lazy evaluation strategy and supports fault tolerance. All intermediate result datasets can be cached explicitly. However, as a C++ dataset, it employs compact object storage rather than the general object storage to reduce serialization and deserialization overhead. Lotus dataset supports primary data types, including int, double, and string. For a string dataset, data are organized into two compact buffers: one as indexes and the other as original characters.

Lotus provides a compute engine for LotusSQL. We add structured dataset abstraction on Lotus and connect it with SQL.

2.2 Calcite

Apache Calcite is an open-source software framework

that provides query processing, optimization, and query language support for multiple backends. It perceives that developers of specialized systems encounter related problems, such as query optimization or the need to support query language (e.g., SQL and its extensions). It also helps minimize the engineering effort required to develop similar optimization logic and language support as a unifying and pluggable framework.

Relational algebra lies at the core of Calcite. Two types of operators express data manipulation operations during the query compilation and optimization process. A logical operator is the primary form of operation, and it includes filter, project, and join. A physical operator assigns an implementation method in an actual backend to a logical operator. Operators compose the relational algebra expression tree, which is the representation of an execution plan. An execution plan consisting primarily of logical operators is called a logical plan. Similarly, an execution plan consisting mainly of physical operators is called a physical plan.

Calcite's query optimization strategy derives from the Volcano[15] and Cascades[16] frameworks. It uses rule-based expression transformation and cost-based dynamic programming search to find the best execution plan.

Calcite's architecture also consists of a query processor capable of processing a variety of query languages, an adapter architecture designed for extensibility, and support for heterogeneous data models and stores (relational, semistructured, streaming, and geospatial). However, these components are not involved in our system and are thus excluded in our discussion.

LotusSQL employs Calcite's optimizer as a frontend to produce a physical execution plan. We supplement Calcite further to deliver excellent results.

3 Workflow

As Fig. 1 illustrates, the processing starting from SQL queries and ending at actual execution results can be divided into four parts. First, an SQL query is taken and parsed into a logical plan with the schema information. Second, a series of optimizations are conducted to generate the physical plan. Third, the physical plan is mapped into C++ codes of structured data operations by the code generator. Finally, the codes are run on Lotus, and corresponding results are obtained. In the following parts of this section, we further describe the process in detail.

3.1 Schema provision and maintenance

In the perspective of SQL queries, data are organized as tables. Tables correspond to a structured datasets in the compute engine, and they share the same logical structure called schema. A schema contains the metadata information of tables, with the most important being the names and data types of columns. This information is the basis for processing SQL queries*.

In LotusSQL, schema acquisition includes two aspects: Calcite schema acquisition and schema acquisition. Calcite schema acquisition is used to complete tasks, such as SQL parsing and optimization. Schema acquisition in compute engine is used to locate tables in a file system and conduct compilation inference. The two schemas must be consistent to ensure the correctness of execution information, such as column indexes.

The schema acquisition in Calcite adopts the method of file parsing. An external file stores relevant information in a specific format, such as CSV. LotusSQL reads and analyzes the file and registers the information into the framework before query parsing. The schema in the C++ program is represented as template arguments. Thus, the schema maintenance in the compute engine is completed in the compilation stage by automatic type inference. LotusSQL ensures that codes delivered to the compute engine are compilable because the parsing and optimization stages also maintain the correct schema. The only extra information that needs to be provided

*The definition of schema here is different from the `Schema` class in Calcite, which is a namespace for a series of tables.

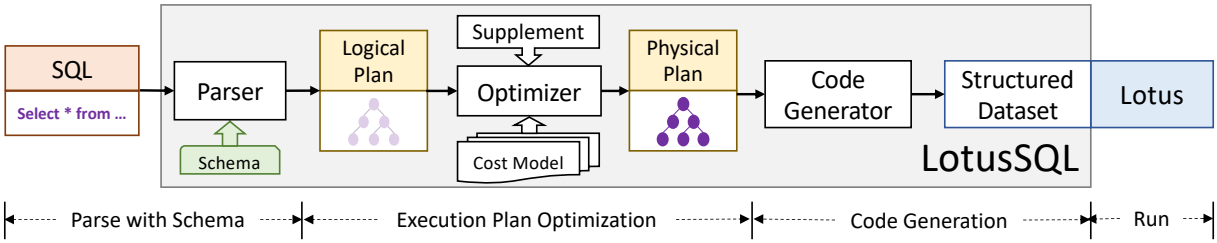


Fig. 1 Workflow overview.

to the compute engine is the storage location of a table. This information can be provided by maintaining relevant information in Calcite's table or by storing and parsing additional files during execution.

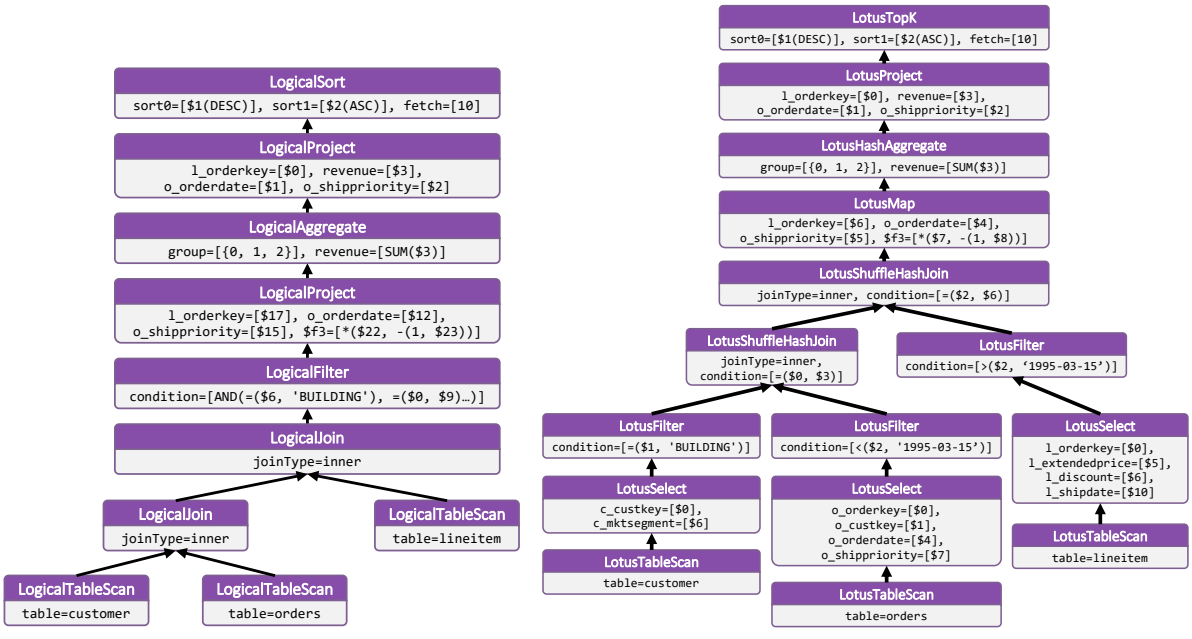
3.2 Execution plan

Calcite's optimizer uses an expression tree of relational operators as its internal representation of an execution plan. In addition to Calcite's built-in logical operators, LotusSQL customizes physical operators corresponding to structured data operations. A physical operator appoints an implementation method of a logical operator. For example, a logical operator LogicalJoin with equality conditions could be implemented by the physical operator LotusBroadcastHashJoin or LotusShuffleHashJoin. We will introduce these operators and operations in Section 4. Each physical operator has a cost function to provide its cost to the optimizer. Cost estimation is related to the specific implementation and relies on some metadata, such as the output number of rows and number of distinct values in a column. Calcite's metadata provider can help to make many deductions, such as row number estimation and unique key tracking. Our supplement to Calcite is presented in Section 5.

An execution plan composed of logical operators is a logical plan, whereas a physical plan comprises physical operators. During compilation, a query is parsed into a logical plan. The logical plan needs to be further transformed into a physical plan, which then determines the implementation of each operator. However, one can decide how to execute a query only if it knows the implementation of every operator. That is to say, a physical plan is needed. Calcite can serve as a mature framework of execution plan transformation. The query optimizer we use in Calcite builds on the ideas from the Volcano[15] and Cascades[16] frameworks, in which the conversion from a logical plan to a physical plan and query optimization are combined.

The optimization is based on rules and the operator cost model. A rule matches a given pattern in the tree and performs a transformation that preserves the semantics of that expression. Calcite includes a set of such rules to transform expression trees. In addition, we define rule conversion as the conversion of logical operators into their corresponding physical operators. All rules are registered in the optimizer. The optimizer creates and tracks the different alternative plans created by firing the rules. By exhaustively exploring the search space until all rules have been applied to all expressions, the original expression tree is expanded to a directed acyclic graph of sets. Each set consists of logical and physical operators with equivalent semantics. With the cost information provided by operators, the optimizer picks up implementations using the dynamic programming algorithm to reduce the overall expression cost.

As an illustrational example, Figs. 2a and 2b depict the logical plan and optimized physical plan of TPC-H Q3. Optimization methods, such as column pruning and



(a) Logical plan (b) Optimized physical plan

Fig. 2 Execution plans for TPC-H Q3.

filter push-down, are applied. The physical plan specifies join implementations.

3.3 Code generation

LotusSQL generates codes according to the physical plan and executes them in the compute engine to obtain the final result of a query. Every physical operator has a code generation function interface. The functions are triggered recursively along the physical plan to generate the codes of all involved dataset operations. Codes are then integrated into a template C++ file and delivered to the compute engine.

Nullable types and values need to be handled carefully in the code generation stage. Nullable values may come from nullable source columns, outer joins, and global aggregations. The compute engine uses `std::optional` to deal with nullable types, and this application leads to the problem of nullable value access not sharing the same pattern as the primitive type. Expressions referring to nullable values can only be produced inside a non-nullable check field.

4 Physical Operators

Physical operators constitute physical plans, which connect the query optimizer and the compute engine through code generation. Physical operators in the optimizer and structured dataset operations in the compute engine show a one-to-one correspondence. This section discusses the physical operators and their corresponding operations.

4.1 Operation fusion

Before delving into the details of operators, we introduce an effective technique to dataset operation implementation: operation fusion. This concept was first proposed in the main-memory database field[17]. In an algebraic expression, operations that do not take incoming tuples out of CPU registers can be fused together to maximize data and code locality. For example, consider a map operation followed by a filter operation. In performing a single map operation, every tuple needs to be loaded from the main memory into the registers. Then, the map expression is calculated, and the output tuple is written back. With regard to the filter operation, similar steps are performed. However, the memory access in-between is unnecessary because the element dependency is determined and can be pipelined. Operation fusion leaves tuples in registers and makes the execution cheap. Note that this scenario does not only happen in individual operators, such as maps and filters. A structured dataset operator may be formed by several basic operations, such as `map` and `shuffle`. Operation fusion can happen between an operator and its neighbor's components.

LotusSQL borrows the philosophy but has a different implementation methodology. Instead of real-time computing and iterator modeling in databases, our compute engine adopts lazy evaluation, such as Spark, by tracking operation dependencies and triggering the required computation. Dependencies exist in two levels: one in the C++ code level and the other in the execution level. Before compilation, the C++ template metaprogramming is used to build the dependency of operations. Each operation invokes a template method with the signature `produce(partition, outConsumer)` to scan through a given input partition and yield the output elements through `outConsumer`, which is a stateful lambda expression that is called for each output element. At the compilation stage, the C++ compiler automatically merges the lambda expressions. This step compresses the original dependencies and forms a compressed one for execution.

`LotusSQL` generates C++ codes by using physical plans, which hold the code level dependencies of queries. However, during optimization in the execution plans, the cost model considers the operation fusion to evaluate the actual running cost.

4.2 Implementation and cost model

Table 1 lists our physical operators (each for an encapsulated dataset operation in the backend) and their corresponding logical operators. The description provides a short introduction to the implementation. The cost model evaluates the costs of the implementation.

Operation fusion has an impact on the cost evaluation of `LotusSelect`. A structured dataset in the compute engine exists as columnar objects when materialized. Hence, selecting columns from the dataset is a lightweight action that only requires copy object pointers. However, if the operation has been fused in the compilation stage, then the selection transforms into an element-wise operation and becomes the same as `LotusMap`. This difference may change the operator order in an execution plan. For example, a `LotusTableScan-LotusFilter-LotusSelect` operator chain is more expensive than `LotusTableScan-LotusSelect-LotusFilter` because a dataset reading from the file system is already

Table 1 Operator list.

LogicalOp	PhysicalOp	Description
TableScan	LotusTableScan	Read a table (dataset) from the file system.
Filter	LotusFilter	Filter a table by given condition.
Project	LotusSelect	Select some columns from a table.
	LotusMap	Map table rows by given expression.
Aggregate	LotusAggregate	Aggregate all rows by given function.
	LotusHash Aggregate	Aggregate rows by given group and function via HashMap.
Join	LotusCartesian Product	Calculate cartesian product of two tables.
	LotusBroadcast HashJoin	Join two tables via broadcasting one to the other and HashMap.
	LotusShuffle HashJoin	Join two tables via re-partitioning tables and using HashMap.
Sort	LotusSort	Sort all rows by given reference key and direction.
	LotusTopK	Find top-k rows by given reference key and direction.

in columnar storage and thus entails a relatively low cost for selection.

Taking the concrete implementation as a reference, we implement the cost function for each physical operator, which comprises the cost model for LotusSQL. Generally, the cost can be depicted in several aspects, such as CPU usage, memory access, and I/O bytes. However, considering additional factors brings few marginal benefits, and forecasting numerous indicators accurately is somehow impossible in the query optimization stage. Thus, in this work, we adopt CPU usage as the only metric for the operation cost.

Take `LotusBroadcastHashJoin` as an example. It is an implementation of join with an equality condition. It is chosen when one table of join operands is small. Its cost consists of several parts.

`LotusBroadcastHashJoin` takes two input tables distributing on the left and right sides in the operator tree. Assuming the left input table is broadcasted to all partitions of the right one during execution, the broadcasting cost thus given by

$$\begin{aligned} Cost_{broadcast} = & LeftInputRowCount \times \\ & LeftInputColumnCount \times \\ & NumRightPartition. \end{aligned}$$

The left table is used to build a HashMap, with the columns involved in the equality condition serving as the hash key. The cost of building and searching the HashMap depends on the entry number of the HashMap. The entry number can be reflected by the metadata DistinctRowCount, which is estimated by the SQL engine. The cost of building and searching the HashMap is

$$\begin{aligned} Cost_{hashMap} = & (LeftInputRowCount \times NumRightPartition + \\ & RightInputRowCount) \times \\ & \log(LeftDistinctRowCount). \end{aligned}$$

With the DistinctRowCount, we can calculate the average entry size. If the hash key is also a unique key of the table, that is, the DistinctRowCount is the same as the LeftInputRowCount, then the AvgHashMapEntrySize should be 1. Otherwise, it can be calculated by

$$\begin{aligned} AvgHashMapEntrySize = & \ LeftInputRowCount / \\ & LeftDistinctRowCount. \end{aligned}$$

If the AvgHashMapEntrySize is not 1, then the HashMap entry should be an iterative type, which entails additional overhead for dynamic memory management. Under this condition, the cost for the HashMap operation updates is

$$Cost_{hashMap} = Cost_{hashMap} \times (AvgHashMapEntrySize + 1).$$

After searching, the concat and output cost is

$$Cost_{output} = OutputRowCount \times OutputColumnCount.$$

The final cost is

$$Cost = Cost_{broadcast} + Cost_{hashMap} + Cost_{output}.$$

5 Query Optimization

Calcite offers a mature framework and many optimization rules, but it should still be supplemented to improve the optimization quality.

5.1 Decorrelation of subqueries

Subqueries are common in SQL statements. Subqueries that do not involve external variables are noncorrelated and are parsed into an independent subtree in the execution plan. The same is not true for correlated subqueries with external references. In the original logical plan, the correlated subquery appears as a `LogicalCorrelate` operator. The operator behaves like a special type of join, but the right input subtree refers to variables from the left input. A straightforward implementation is that for each tuple from the left input, the right subtree is executed, and the union of all the outputs obtained is taken. However, re-executing the right subtree every time hampers performance in

most cases. Thus, decorrelation is necessary, that is, the `LogicalCorrelate` operator should be removed before entering rule- and cost-based optimization.

Calcite adopts several methods for decorrelation, but they are not efficient enough. Figure 3 illustrates the decorrelation process of Calcite. Some rules match simple patterns for quickly removing `LogicalCorrelate`. However, for complex subqueries, the operator tree should be decorrelated recursively. Before and after recursive decorrelation, rules are also applied. The post-rules can be regarded as a part of the subsequent optimization. Thus, we do not discuss them in this paper. The pre-rules include aggregation adjusting, filtering and projection transposing, and condition push-down. The main purpose is to ensure that the operators are in fixed patterns and thus simplify the decorrelation procedure. For example, after applying the `FilterProjectTranspose` rule, the order in a `Filter` and `Project` operator pair is fixed. In the default implementation of Calcite, each pre-rule is applied once. However, such application is not enough because of the recursive decorrelation procedure.

In the recursive decorrelation procedure, subtrees need to be copied in certain cases. Figure 4 shows an example. The left side is a part of a logical execution plan, in which the filter condition of the `LogicalFilter` operator contains correlated variables. In this case, the left subtree (SubTree_1) needs to be copied, and the `ValueGenerator` operator (which is responsible for generating all the values of the referred correlated variables) needs to be added to join with the original input operator. Consider this local execution plan as the input of another `LogicalFilter` operator, which contains filtering conditions that can be pushed down to SubTree_1. If the condition is not pushed into SubTree_1 before decorrelation, then the copied subtree existing in the right subtree of `LogicalJoin` would never meet the condition.

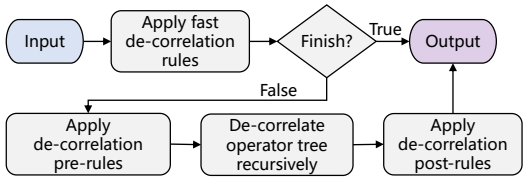


Fig. 3 Calcite decorrelation.

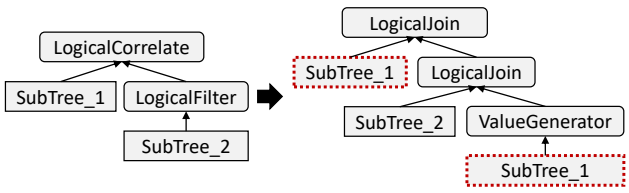


Fig. 4 Decorrelation example.

The preprocedure of decorrelation should push the optimization down to the internal subtree as much as possible. If SubTree_1 already contains all the information needed for optimization, then the optimization can be realized through the subsequent optimization process after copying. To achieve this goal, we add the update check after the preprocedure and apply the pre-rules repeatedly so that Calcite enters the process of recursive decorrelation only when the execution plan does not change. In this way, the query performance under the condition in which multiple complex subqueries coexist is improved considerably.

5.2 Condition expression transformation

Condition push-down is a natural rule for reducing the dataset scale as early as possible. However, the condition form matters when performing the rule. We add a condition expression transformation rule to Calcite. In some cases, this step can help the condition to be pushed down deeply.

At the beginning of query optimization, conditions exist only in `LogicalFilter` operators. If the input of a filter operator is a single-input operator, such as a map or aggregation, then condition push-down can always be accomplished by exchanging the order of two operators and carefully adjusting the index number in the condition. However, when the input is a join, the condition is merged into the input operator and whether it can be pushed down further is subsequently determined. Figure 2a shows that a plain `LogicalJoin`'s semantic is a Cartesian product. After a condition is pushed into the join operator, it analyzes the condition to try to acquire the equality condition for the physical operation and then pushes down the other conditions.

To understand the condition push-down in join, we consider the condition from the perspective of Boolean algebra. A condition is composed of unit conditions and basic operations. The basic operations of Boolean algebra are as follows: AND ($\wedge$), OR ($\vee$), and NOT ($\neg$). Unit conditions are specific simple conditions, such as range check and regular expression matching check that do not include logical operations. When analyzing a condition expression in a join, conjunction can be decomposed. For a condition $C_1 \wedge C_2 \wedge \cdots \wedge C_n$, subexpressions C_i can be divided into three sets:

- Subexpressions that refer to columns from only one

input. They can be pushed down into the corresponding input side;

• Subexpressions that refer to columns from both inputs and with equality conditions. They remain in the join operator to avoid the Cartesian product calculation;

• Subexpressions that refer to columns from both inputs and with non-equality conditions. They are applied to the output of join.

To ensure execution efficiency from the first two cases, we define a new rule that transforms a disjunction to an equivalent conjunction. The transformation involves two steps: (1) to move the negation operation inward to the unit condition according to De Morgan's laws[18] and (2) to try to extract common unit conditions from disjuncts and make them a conjunction if the entire expression is a disjunction.

Note that the transformation is not aimed toward a conjunctive normal form. Conjuncts in the conjunctive normal form usually involve multiple unit conditions and, thus, multiple input data columns. They are not conducive to decoupling and push-down. Compared with complex propositions in common Boolean algebra, conditions appearing in SQL queries are usually less complicated. Thus, this one-step transformation is a reasonable choice to balance the push-down effect.

Take TPC-H Q19 as an example. The query is to filter and aggregate the results of a single join. The condition is at first pushed into the join, and its transformation can be expressed as Eq. (1)†:

$$\begin{aligned} Conditon = {} & (LRE_1 \wedge L_1 \wedge L_2 \wedge L_3 \wedge R_1 \wedge R_2 \wedge R_3) \vee \\ & (LRE_1 \wedge L_4 \wedge L_5 \wedge L_6 \wedge R_4 \wedge R_2 \wedge R_3) \vee \\ & (LRE_1 \wedge L_7 \wedge L_8 \wedge L_9 \wedge R_5 \wedge R_2 \wedge R_3) = \\ & LRE_1 \wedge R_2 \wedge R_3 \wedge ((L_1 \wedge L_2 \wedge L_3 \wedge R_1) \vee \\ & (L_4 \wedge L_5 \wedge L_6 \wedge R_4) \vee (L_7 \wedge L_8 \wedge L_9 \wedge R_5)) \end{aligned} \quad (1)$$

The condition before the transformation cannot be pushed down because it involves columns from both sides and is a disjunction. After the transformation, LRE_1 can be used as the equivalent condition of the join operation, while R_2 and R_3 can be pushed down to the right operator to reduce the amount of input data.

5.3 RowCount estimation

Calcite users can invoke `getRowCount()` to estimate the number of output rows of an operator. The estimation is based on Calcite's mechanism that provides metadata. The mechanism can help to estimate information such as RowCount and condition selectivity. It also tracks inherited properties, such as unique keys and column origins. Herein, we introduce our work that utilizes unique key information to estimate the accurate RowCount of join.

A unique key is a group of one or more fields or columns of a table that uniquely identify row records. Consider the estimation of the output RowCount of the simple query `select*from TableA, TableB` where `TableA.x=TableB.y`. In the default implementation of Calcite, its estimated number of rows is

$$\begin{aligned} RowCount = {} & TableA.RowCount \times TableB.RowCount \times \\ & Selectivity(TableA.x = TableB.y) \end{aligned} \quad (2)$$

`Selectivity` is a simple guess that returns a value between 0.5 and 1.0. The guess does not involve any table-related information and returns the same value for all equality check. However, for each row in TableB, if TableA.x is assumed to be a unique key for TableA, then a maximum of one row in TableA can be found to meet the condition `TableA.x=TableB.y`. Therefore, we can deduce that the RowCount should not exceed TableB. RowCount. It should be much less than the default estimation in most cases. For other types of join operations, the upper bound of the number of rows can also be calculated. Under the assumption that the columns involved in the left table in the equality conditions constitute a unique key, Table 2 lists the upper bounds of the estimated number of rows for different types of join operations.

The column uniqueness information can be offered when providing a schema for source tables. The metadata provider tracks the information automatically when such queries are invoked.

The estimation of RowCount is the basis of the cost model. The join operator is almost the most expensive of all operators and exerts great impact on the overall cost of an entire query. Our estimation yields highly accurate results and thus helps produce good execution plans.

6 Evaluation

In this section, we first analyze the query translation quality of LotusSQL. To assess the performance and acceleration potential of LotusSQL relative to Spark

Table 2 RowCount estimation for join.

Join Type	RowCount upper bound
InnerJoin, RightJoin	`RTable.RowCount`
OuterJoin, LeftJoin	`LTable.RowCount+RTable.RowCount-1`
SemiJoin, AntiJoin	`LTable.RowCount`

† $L_i(R_i)$ represents a unit condition that only refers to the left (right) input; and LRE_I represents an equality unit condition that refers to both inputs.

SQL, we present two experiments on the standard relational benchmark TPC-H[19]. The first experiment involves the comparison of computing times while the second one involves memory usage measurement.

6.1 Workloads and environment

We use the standard TPC-H benchmark for the analysis and experiments. Datasets with a scale factor of 10 (SF10, i.e., 10 GB in total) and 100 (SF100, i.e., 100 GB in total) are used for the running time experiment. Those with a scale factor of 100 are used for the memory usage experiment.

We conduct our experiments on a single machine, the configurations of which are shown in Table 3. The configurations of Spark SQL are presented in Table 4. As the experiment is conducted on a single machine, we deploy Spark SQL in the standalone mode and fine-tune it in terms of memory and computing resource usage.

6.2 Query translation analysis

Optimized physical plans are essential for satisfying query performance. Figure 2b shows the translation result of TPC-H Q3 with SF100. As a relatively simple query involving three tables, the physical plan is nearly the same as a human-optimized one. As for the other queries, especially for those that need more joins and aggregations, LotusSQL also generates satisfactory results. Complex decisions, such as join reordering, are difficult for humans to handle. Meanwhile, LotusSQL performs well in such tasks.

Through the operation abstraction, the codes generated by an execution plan are readable. These codes are cumbersome to write, especially when manually maintaining the index of columns. Remembering the index after several joins takes significant human effort.

Although LotusSQL yields satisfactory plans and achieves good usability, we admit that its query optimization may still be improved. This potential lies in the transformation of subqueries. As shown in Fig. 5, even with a C++ computing engine, LotusSQL behaves relatively slow on some queries. Typically, Q18, Q21, and Q22 all include subqueries leading to

Table 3 Experiment environment.

Item	Discription
CPU	Intel(R) Xeon(R) E5-2680 v4
Frequency	2.40 GHz
Pyhsical Cores	28
Virtual Cores	56
NUMA Nodes	2
Operating System	Ubuntu 16.04.10
Main Memory	512 GB
Disk	6 TB NVMe SSD

Table 4 Spark configuration.

Item	Discription
Spark Version	3.0.1
Hadoop Version	2.7
Java Version	11.0.9
Scala Version	2.12.10
Executors	8
Executor Cores	7

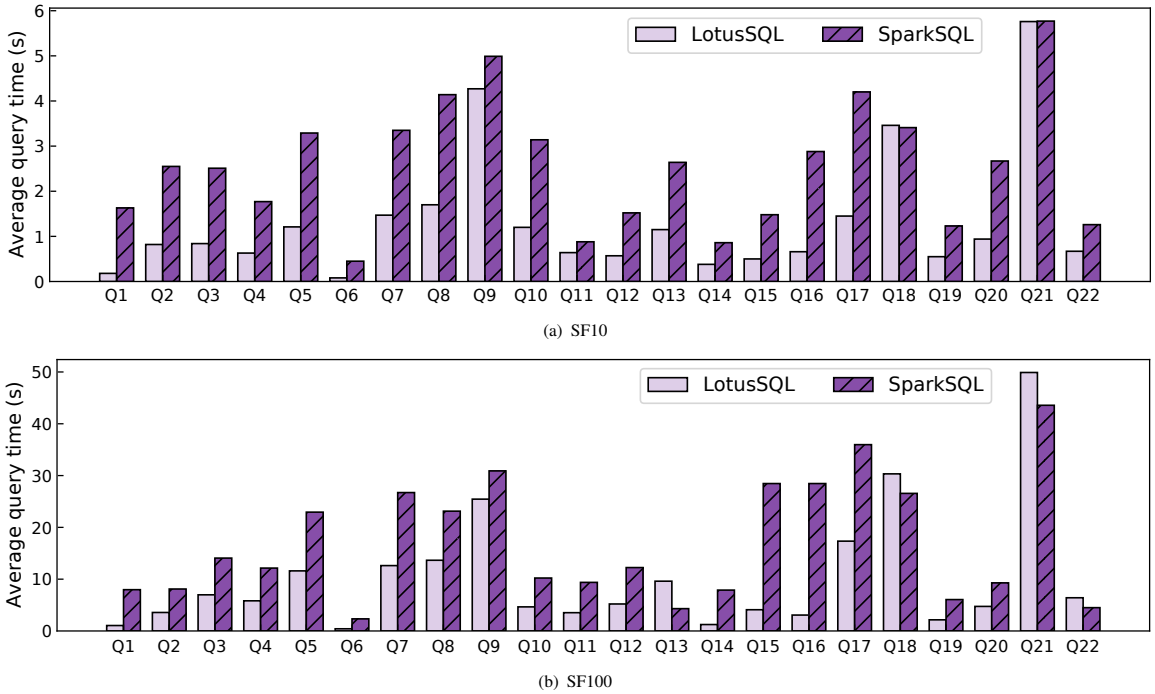


Fig. 5 TPC-H computing time.

expensive joins, which degrade performance. In addition, operation-level optimization, such as dataset indexing, can help improve join performance.

6.3 Computing time

In this experiment, we compare the absolute query computing times of LotusSQL and Spark SQL on a single machine using the TPC-H benchmark with SF10 and SF100. We load all data into the main memory before querying to eliminate the effect of data reading. We use persist() to cache the data in both systems and ensure that they have sufficient memory during computing. We run the query once as a warm-up, repeat it 10 times, and report the average running time as the result.

Figures 5a and 5b show respectively the average computing times of LotusSQL and Spark SQL with scale SF10 and SF100. LotusSQL performs better than Spark SQL in most queries. Across all 22 queries, the geometric average speedups of LotusSQL over SparkSQL for SF10 and SF100 are 2.61× and 2.25×, respectively.

6.4 Memory usage

We perform a memory usage experiment on the dataset with SF100. The experiment evaluates the performance of Spark SQL under different memory constraints. For LotusSQL, we monitor the maximum memory usage for each query. Figure 6 summarizes all queries for both systems in one graph to reflect the overall situation. The dots indicate the performance of Spark SQL under

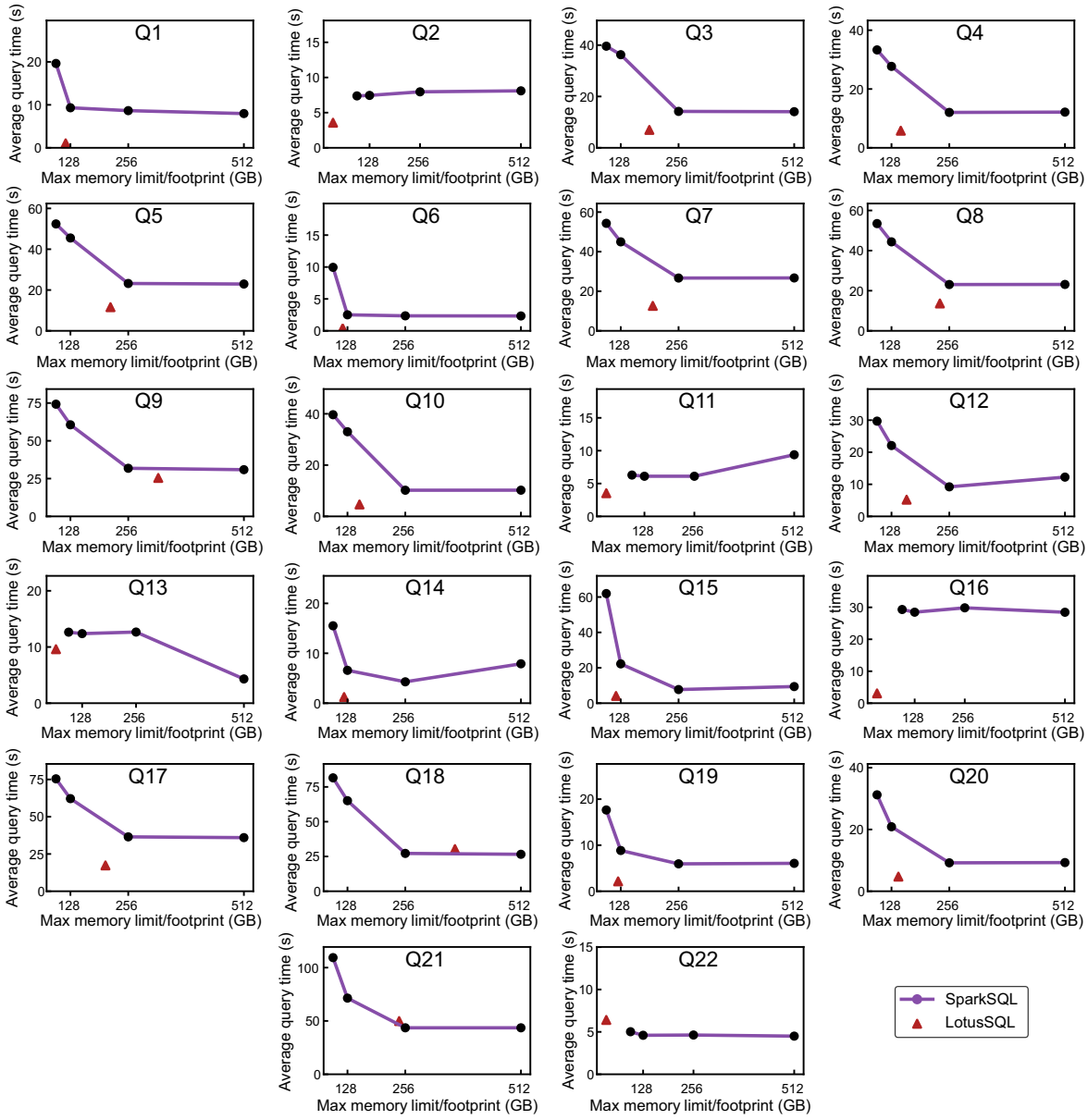


Fig. 6 TPC-H memory usage.

different memory limits (a line mainly identifies the data points of the same query, and it does not represent the performance in a continuous state). The triangle points indicate the maximum memory usage of LotusSQL; they are accurate values with a maximum available memory of 512 GB. We do not mark query IDs as triangles for clarity.

We evaluate the average query time of Spark SQL when the maximum memory for the whole Spark cluster is limited to 96, 128, 256, and 512 GB. Spark SQL involves external storage in data shuffling when the memory is insufficient. The performance degradation caused by external storage access mostly happens before 256 GB. We also try to limit Spark SQL memory usage to 64 GB. However, many queries fail because of insufficient memory under this constraint. Thus, we discard this configuration. The comparison of each query is presented in Fig. 6.

The average maximum memory usage of 22 queries is 153.60 GB, which is about 1.5× the original tables. But for some complex queries, the memory usage is still relatively large, which is reflected by a red triangle appearing at the right side of a sub-figure, e.g., the sub-figures of Q9 and Q18. Lotus adopts a partition-wise computing model that each thread loads a partition into memory at a time. Therefore, re-parting tables into smaller pieces can save memory and make it possible to run on a machine with less memory.

7 Discussion

By observing existing big data processing systems that support SQL and how they support SQL, we can find that they basically follow a similar route, which addresses the challenges mentioned in Section 1. In this section, we discuss our choice and hope that our work can bring some inspiration to this type of problem.

7.1 Calcite as front end

When adding an SQL engine for a system, building it from scratch is not wise. As SQL is such a mature query language, many existing frameworks or modules can facilitate compilation or optimization. Among all these previous works, we choose Calcite as our frontend to translate SQL queries into execution plans.

Adopting Calcite has advantages in several aspects. First, Calcite is a united framework that integrates a query parser and a query optimizer. Thus, we do not need to deal with query parsing and optimization separately. Second, the built-in optimization logic is powerful. For example, Calcite can perform join reordering with the information from a cost model. It is decisive with regard to the performance of some queries, but it is not supported in Spark SQL's Catalyst optimizer. Third, Calcite's modular and extensible architecture makes the addition of new features easy and allows implementation to be adjusted according to our specific backend. Calcite reduces the labor cost needed to develop an SQL engine to an acceptable extent for one person.

Employing Calcite also brings problems that need to be addressed. Its complicated design makes it a heavy project. The source code in the core package of Calcite exceeds 180 000 lines. Understanding this foundation and using it correctly are inevitable challenges. Furthermore, modifying Calcite under the constraints of its architecture requires sophisticated efforts.

7.2 SQL support in system hierarchy

Although LotusSQL sets itself as an SQL module for the Lotus system, it is actually for Lotus' operation API with a cost model. Figure 7 abstracts our system into two layers with a total of five parts. The intermediate parts with purple shadow have direct dependencies while the parser and bottom module can be replaced by any compatible substitute. Calcite leaves storage management to the backend and provides Adapter to connect to them. LotusSQL does not adopt the adapter design, but it can support specialized low-level implementation. Lotus can replace the in-memory storage module with a distributed one transparently. We will analyze the distributed version in the future. The change of the bottom module may require an adjustment of the cost model. For Lotus, such adjustment is not needed because our cost model is not that fine-grained and our physical operations keep the relative cost in distributed mode.

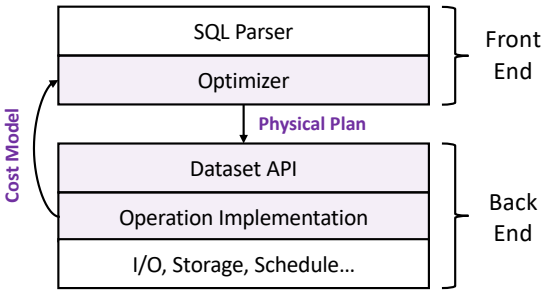


Fig. 7 System hierarchy.

8 Related Work

8.1 SQL for big data systems

Since the introduction of the Google File System[20] and MapReduce model[21], big data processing based on their ideas has grown rapidly. Hadoop[1] and its file system[22] support fault-tolerant MapReduce workloads on large clusters. Twister[2] and Haloop[3] improve iterative MapReduce workloads by keeping invariant data between iterations. Spark[4] further supports persisting data in memory to enhance data reuse. Another contribution of Spark is the abstraction RDD and its concise interfaces.

On top of these systems, the need for OLAP on structured and semistructured data encourages many specific modules or subsystems. Pig[23], Hive[24], and Impala[25] are upper-level query engines for Hadoop supporting SQL or SQL-like language. Pig and Hive are developed in Java and integrated with native Hadoop while Impala uses C++ and does not build on MapReduce. However, Impala lacks some important features, such as correlated subqueries and custom user-defined functions. Shark[26] and Spark SQL[7] are SQL engines for Spark. Spark SQL adopts a completely new query optimizer and is widely used because of its complete functionality and energetic community.

Asterix[27] and Stratosphere[28] are other systems that overtake Hadoop or Hive in various dimensions. Tupleware[29] is a system that is explicitly aimed towards complex analytics on small clusters. Structured Computations Optimized for Parallel Execution (SCOPE)[30] is a structured data processing language presented by Microsoft. SCOPE has its particular distributed compute engine.

Although these systems are impressive, LotusSQL sets itself apart by building an SQL engine on general dataset abstraction written in C++ and achieving relational performance improvement over Spark SQL by more than 2× on average on a full set of TPC-H queries.

8.2 Query compilation

The iterator model for query evaluation is classical in relational databases. It was proposed relatively early[31]. Volcano[15] and Cascades[16] are the most commonly used optimization strategies today, as they are flexible and quite simple.

The query compilation strategy[17] proposed by Thomas Neumann on HyPer[32] inspires LotusSQL. Whereas HyPer is a main-memory database and translates queries into machine code directly, LotusSQL is based on a big data processing system and generates C++ codes.

8.3 Performance evaluation

McSherry et al.[33] proposed the Configuration that Outperforms a Single Thread (COST) metric and showed that in many cases, single-threaded programs could outperform big data processing frameworks running on large clusters. This finding reminds us that cumbersome clusters may not be the best and only choice for big data processing. We agree with such notion given the fact that high-bandwidth out-of-core storage is becoming the new mainstream.

TPC-H[19] is a commercial decision support benchmark that consists of 22 analytical queries. Almost all typical query patterns, including aggregation, large join, sorting, and correlated subqueries, can be found in TPC-H.

9 Conclusion

Modern data analytics need to make efficient use of modern hardware with large memory and numerous cores. Lotus is such a general big data processing system developed with native programming language. To boost Lotus with a convenient and expressive user interface, we present LotusSQL. LotusSQL uses Calcite to compile and optimize queries with the guidance of a physical cost model. Structured datasets are designed to be compatible with native Lotus datasets but adopt columnar storage. The dependencies are resolved as a whole and compressed during C++ compilation time. With all these strategies, LotusSQL outperforms Spark SQL in TPC-H queries by more than twice on average.

In future LotusSQL versions, we plan to support prepared statements in SQL queries to enable query plan reuse, and to improve the optimization toward correlated subqueries.

References

[1] Apache Hadoop, Apache hadoop, http://hadoop.apache.org, 2021.

[2] J. Ekanayake, H. Li, B. J. Zhang, T. Gunarathne, S. H. Bae, J. Qiu, and G. Fox, Twister: A runtime for iterative mapreduce, in *Proc. 19th ACM Int. Symp. on High Performance Distributed Computing*, Chicago, IL, USA, 2010, pp. 810–818.

[3] Y. Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, HaLoop: Efficient iterative data processing on large clusters, *Proc. VLDB Endowm.*, vol. 3, nos. 1&2, pp. 285–296, 2010.

[4] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M.

McCauly, M. J. Franklin, S. Shenker, and I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in *Proc. 9th USENIX Conf. on Networked Systems Design and Implementation*, Berkeley, CA, USA, 2012, pp. 15–28.

[5] F. Yang, J. F. Li, and J. Cheng, Husky: Towards a more efficient and expressive distributed computing framework, *Proc. VLDB Endowm.*, vol. 9, no. 5, pp. 420–431, 2016.

[6] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, Apache flink^{TM}: Stream and batch processing in a single engine, *Bull. IEEE Comput. Soc. Tech. Comm. Data Eng.*, vol. 36, no. 4, pp. 28–38, 2015.

[7] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. R. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al., SparkSQL: Relational data processing in spark, in *Proc. 2015 ACM SIGMOD Int. Conf. on Management of Data*, Victoria, Australia, 2015, pp. 1383–1394.

[8] M. Anderson, S. Smith, N. Sundaram, M. Capot? Z. G. Zhao, S. Dulloor, N. Satish, and T. L. Willke, Bridging the gap between HPC and big data frameworks, *Proc. VLDB Endowme.*, vol. 10, no. 8, pp. 901–912, 2017.

[9] G. M. Essertel, R. Y. Tahboub, J. M. Decker, K. J. Brown, K. Olukotun, and T. Rompf, Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data, in *Proc. of the 13th USENIX Conf. on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2018, pp. 799–815.

[10] L. Lu, X. H. Shi, Y. L. Zhou, X. Zhang, H. Jin, C. Pei, L. G. He, and Y. Z. Geng, Lifetime-based memory management for distributed data processing systems, *Proc. VLDB Endowm.*, vol. 9, no. 12, pp. 936–947, 2016.

[11] C. Navasca, C. Cai, K. Nguyen, B. Demsky, S. Lu, M. Kim, and G. H. Xu, Gerenuk: Thin computation over big native data using speculative program transformation, in *Proc. 27th ACM Symp. on Operating Systems Principles*, Ontario, Canada, 2019, pp. 538–553.

[12] J. Arnold, B. Glavic, and I. Raicu, A high-performance distributed relational database system for scalable OLAP processing, in *2019 IEEE Int. Parallel and Distributed Processing Symp.* (*IPDPS*), Rio de Janeiro, Brazil, 2019, pp. 738–748.

[13] T. Bingmann, M. Axtmann, E. Jöbstl, S. Lamm, H. C. Nguyen, A. Noe, S. Schlag, M. Stumpp, T. Sturm, and P. Sanders, Thrill: High-performance algorithmic distributed batch data processing with C++, in *2016 IEEE Int. Conf. on Big Data* (*Big Data*), Washington, DC, USA, 2016, pp. 172–183.

[14] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire, Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources, in *Proc. 2018 Int. Conf. on Management of Data*, Houston, TX, USA, 2018, pp. 221–230.

[15] G. Graefe, and W. J. McKenna, The volcano optimizer generator: extensibility and efficient search, in *Proc. IEEE 9th Int. Conf. on Data Engineering*, 1993, Vienna, Austria, pp. 209–218.

[16] G. Graefe, The cascades framework for query optimization, *Data Eng. Bull.*, vol. 18, no. 3, pp. 19–29, 1995.

[17] T. Neumann, Efficiently compiling efficient query plans for modern hardware, *Proc. VLDB Endowm.*, vol. 4, no. 9, pp. 539–550, 2011.

[18] Wikipedia, De Morgan's, https://en.wikipedia.org/wiki/De Morgan%27s, 2021.

[19] The Transaction Processing Performance Council, TPC-H vesion 2 and version 3, http://www.tpc.org/tpch/, 2021.

[20] S. Ghemawat, H. Gobioff, and S. T. Leung, The Google file system, in *Proc. 19th ACM Symp. on Operating Systems Principles*, Bolton Landing, NY, USA, 2003, pp. 29–43.

[21] J. Dean and S. Ghemawat, MapReduce: Simplified data processing on large clusters, in *6th Symp. on Operating System Design and Implementation* (*OSDI 2004*), San Francisco, CA, USA, 2004, pp. 137–150.

[22] K. Shvachko, H. R. Kuang, S. Radia, and R. Chansler, The Hadoop distributed file system, in *2010 IEEE 26th Symp. on Mass Storage Systems and Technologies* (*Msst*), Incline Village, NV, USA, 2010, pp. 1–10.

[23] C. Swarna and Z. Ansari, Apache pig–A data flow framework based on Hadoop map reduce, *IJETT J.*, vol. 50, no. 5, pp. 271–275, 2017.

[24] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, Hive–A petabyte scale data warehouse using Hadoop, in *2010 IEEE 26th Int. Conf. on Data Engineering* (*ICDE 2010*), Long Beach, CA, USA, 2010, pp. 996–1005.

[25] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, et al., Impala: A modern, open-source SQL engine for Hadoop, presented at 7th Biennial Conf. on Innovative Data Systems Research (CIDR'15), Asilomar, CA, USA, 2015.

[26] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, Shark: SQL and rich analytics at scale, in *Proc. 2013 ACM SIGMOD Int. Conf. on Management of Data*, New York, NY, USA, 2013, pp. 13–24.

[27] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras, ASTERIX: Towards a scalable, semistructured data platform for evolving-world models, *Distrib. Parallel Databases*, vol. 29, no. 3, pp. 185–216, 2011.

[28] A. Alexandrov, R. Bergmann, S. Ewen, J. C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al., The stratosphere platform for big data analytics, *VLDB J.*, vol. 23, no. 6, pp. 939–964, 2014.

[29] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Cetintemel, and S. Zdonik, Tupleware: "Big" Data, Big Analytics, Small Clusters, presented at 7th Biennial Conf. on Innovative Data Systems Research (CIDR 2015), Asilomar, CA, USA, 2015.

[30] R. Chaiken, B. Jenkins, P. Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. R. Zhou, SCOPE: Easy and efficient parallel processing of massive data sets, *Proc. VLDB Endowm.*, vol. 1, no. 2, pp. 1265–1276, 2008.

[31] R. A. Lorie, *XRM-An Extended* (*N-ary*) *Relational Memory*. Yorktown Heights: IBM, 1974.

[32] A. Kemper and T. Neumann, HyPer: A hybrid OLTP&OLAP main memory database system based on

virtual memory snapshots, in *2011 IEEE 27th Int. Conf. on Data Engineering*, Hannover, Germany, 2011, pp. 195–206.

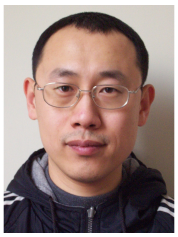
[33] F. McSherry, M. Isard, and D. G. Murray, Scalability! But at what COST? presented at 15th Workshop on Hot Topics in Operating Systems (HotOS XV), Kartause Ittingen, Switzerland, 2015.

Xiaohan Li received the bachelor degree from Tsinghua University in 2018. She is currently pursuing the master degree at Tsinghua University. Her research interests include parallel computing and big data systems.

Bowen Yu received the bachelor degree from Northwestern Polytechnical University in 2015. He is currently pursuing the PhD degree at Tsinghua University. His research focuses on big data systems.

Guanyu Feng received the bachelor degree from Tsinghua University in 2018. He is currently pursuing the PhD degree at Tsinghua University. His research interests include graph processing, graph database, and streaming system.

Haojie Wang received the bachelor degree from Tsinghua University in 2015. He is currently pursuing the PhD degree at Tsinghua University. His research interests include compiler, program analysis, and AI compiler.



Wenguang Chen currently is a professor at Tsinghua University. He received the bachelor and PhD degrees in computer science from Tsinghua University in 1995 and 2000, respectively. His research focuses on parallel and distributed systems and programming systems.