**QPModel Implementation Notes**

**QPModel Team  
10/2020**

# Overview

The world generates data every moment, so it is necessary to solve the two basic problems of data access and analysis. The first is that a file system can store data. Later, the file system developed ISAM, which provided index search capabilities and solved the problem of simple search. IBM developed IMS as the prototype of the database in the 1960s. In the 1970s and 1980s, the development of the two theoretical pillars of the relational model and the transaction model standardized data query methods and transaction semantics, resulting in database products. As the basic product of IT, database has enjoyed a wonderful time for 20 to 30 years.

In the practice of large-scale Internet companies, it is found that (1) the relational model and SQL cannot model and facilitate the analysis of all businesses; (2) the transaction model is too tightly managed, and the performance not scale well. Therefore, the requirements of source code customization, cost, query transaction and data flexibility required by its business make traditional database vendors unable to respond in time, which has given birth to various implementation solutions such as NoSQL, sharding, and special databases:

* HDFS sacrifices the update capability of the POSIX file system in exchange for its storage scalability. HDFS has become the foundation of large-scale data storage, and on top of this, file formats (such as Parquet) and databases based on it have been born.
* MapReduce implementations (such as Hadoop, Spark) provide API programming methods that are more flexible than SQL, while the PL/SQL system tries to build all programming capabilities. Spark goes a step further. On the one hand, it provides SQL, but also provides deep language integration capabilities.
* NoSQL (such as Hbase, AWS DynamoDB, C\*) provide a relaxed transaction model than ACID. This flexibility provides correctness in exchange for the scalability of the system. Sharding stores data in multiple independent databases. Although it cannot provide the semantic guarantee of a single database, it is effective in scenarios where these semantics are not required (for example, there is no cross-shard operation).
* Graph databases (such as Neo4j) provide the ability to directly manipulate graph models, which is much more efficient than simulating the parent-child relationship on a relational system. Similar special databases include memory databases, time series databases, blockchain databases, and so on. Streaming database is different from other models, it is for dynamic data.

## An Ideal Database

For the criticism of the relational model and the transaction model, our thoughts are:

* The database system needs relational model and SQL, but it must be able to be extended and compatible with other models to provide enough programming flexibility. Expansion should solve the two basic problems of access and query (1) Query: Spark's DataSet API, CQL and PGQL are used as extensions of SQL. (2) Access: Specialize storage by introducing new tables and indexes. This scalability provides multiple copies of data consistency and zero delay from a data perspective; from the perspective of engineering development, it emphasizes code reuse and avoids repeated investment. To understand why this is feasible, consider the implementation of Cube, R-Tree, and full-text indexing in the database.
* The database system needs a transaction model, otherwise in theory we cannot guarantee the correctness of the data. From the perspective of isolation level and performance balance, the transaction model needs to consider the difference between stand-alone, local cluster and Geo-cluster, and achieve the ultimate performance and scalability within the theoretically permitted range. The database should also be able to provide a loose transaction model to meet the need for flexibility.

Therefore, we need an extensible modern relational database. If we further examine it with its potential users:

* DBA/cloud service provider: They are concerned about not having problems with the database, TCO, not complicated management, and being able to scale with workload (low cost), etc.;
* Database application developers: They are concerned about SQL compatibility, as well as easier-to-use interfaces such as DataFrame, the convenience of development tools, stable query performance, and ensuring the semantics of the database;
* Data scientists: What they care about is that the database can provide easier operation and can seamlessly connect with the surrounding development environment. Don't spend time on data wrangling.
* Database kernel and tool developers: It is relatively easy to debug and extend functions to the database kernel. Development tools can easily obtain the internal information of the database.
* Other data platform developers: They hope not to repeat the work, if some parts of the database can be reused, they can reduce a lot of repeated development.

We summarize above as REPO:

* **R**esilient: no worry about data loss and service interruption. (1) When the back-end server fails, the database user's interruption perception is at the second level[[1]](#footnote-1); (2) Another type of unavailability that is easy to ignore comes from the implementation of the database itself, because whether it is the user sending DDL It is still doing maintenance work in the background, and the service can hardly be downgraded; (3) The database can be scale out with the expansion of the business; (4) The exception is that if a service and copy are not online, then the system cannot be online.
* **E**ase of Use: most of the time, user input is not required for business-unrelated things (such as tuning). Database tuning is an important work for DBA/developers on different levels: DBAs may tune the configurations and developers may tune parameters for plans. Recent advances in AI may help in the area.
* **P**erforming**:** Influenced by two theories: physical limit and Amdahl’s law, we cannot infinitely improve performance by expanding computing resources. So how to measure reaching the limit? (1) The general code efficiency of the executor and the efficiency of matching handwritten special code, make full use of multi-processors and contemporary hardware, including SCM/RDMA, to ensure that the leading benchmark performance (C/E/H/DS) is maintained; (2) Geo-deployable provides a single global database view, which is added as an expansion capability on the basis of almost no loss of local cluster performance; (3) performance stability.
* **O**pen to extend to arbitrary data and popular data models. Arbitrary data also means unlimited capacity. This requires multi-machine storage and execution capabilities that are transparent to users. The multi-machine cluster system still needs to transparently provide users with a single ACID database view, which is better than the sharding solution. Arbitrary data also means mixed load, multi-mode capabilities, including access and query capabilities for TP, AP, streaming data, and graph models.

To satisfy the REPO requirements, here is a summary of technologies used are here (bold ones are specific to OLTP):

|  |  |
| --- | --- |
| **Component** | **Technology Vocabulary** |
| Interface | * SQL [dialects], DataFrame|Pandas, LINQ, Multi-modal, Language integration, Information schema, error correction, intellisense, tooling, Spark * ORM vertical integration * Extension point APIs, componentization, word counting, pi computation |
| Optimizer | * Cost based staging * Plan versioning/guide * SQL, DataFrame, udf, full program optimization, top down/bottom up, classic operators, external dumb|smart data sources, recommendations, advanced stats, parameter queries, QPModel * Multi-modal, tpc\*, jobench, data lake queries, ML based CE/Costing, algerbrizer|optimizer|\* rule injection, subplan resolver |
| Executor | * Staged Exec, resumable * Cliff-avoidance, approximation, adaptive, always on monitoring, provenance, RM, SQLVM, data privacy, i18n * **CodeGen**, vectorization, **continuation**, HTAP workload, parallel|distributed exec, IVM, CCIX|Arm64, DPU(1~2%)|RISC-V, 1cpu-sec/GB/H, tpc\* * External function, Streaming, multi-modal inter-op, extensible framework |
| Storage Engine | * ACID, independent meta data, **cross-container transaction** * Schema relaxation, TDE, indexing (cube, filtered, etc), **online DDL|resharding**, cloud-integration, **many cores, local or geo-cluster**, **CC flexibility**, partition tables, contented update, CDC, **flashback** * Active storage, NDP, modern HW (esp. **NVM**), **Molehill,** 9\*9 durability, **1sec/recovery, 1M/tpsc/x-core**, **active replica**, Tpc\* * External table, external indexing (with transaction) |

We will explain them in details.

## Fundamentals

Overall logic architecture of a database kernel:

SQL dialects

APIs

Optimizer

Executor

c-AST

c-AST

Storage Engine

c-AST

The system shall be developed with modularization as above. This is good for engineering practice, but also it flexible to re-purpose or enhance them for different scenarios physically. There are four major modular illustrated above:

* The frontend interface modular can be SQL dialects or APIs and they are generated canonical form of ASTs and passed down to the optimizer. This is covered in section 2.
* The optimizer modular processes the AST and generate an executable form of AST and pass down to the executor. This is exercised in QPModel C# and covered in section 3.
* The executor modular executes the AST and some of them are processed in storage engine. Part of the key technologies are exercised in QPModel C# and is covered in section 4.
* The storage engine modular process a subset of ASTs only dedicated for it. This is covered section 5.

Meanwhile, as it is an open source project, we will use public benchmarks TPC\* to drive the technical specifications (like what data types support, maximal capacity, etc) which we don’t specify here. For better development experience, it shall design for debuggability, testability and supportability.

### Software Environment

It relies on cloud storage providing online and offline highly available cross-AZ (but not cross-Region) reliable storage methods. We build the database on cloud storage and don't implements any replication protocols by ourselves. Using cloud storage foundation, database implementation simplifies the following tasks:

* Cloud storage is separately from computing with built-in cross-AZ replication provides high reliability: it shall maintain multiple strong consistent copies with one atomic, async write.
* Cloud storage shall allow limited database logics running on storage servers to move computing closer to the storage, so database can perform operations (say SARG) on storage server for performance considerations.
* Optimize SCM/power-saving memory: cloud storage provides canonical interface on SCM/power-saving memory.
* Cloud storage shall provide a canonical interface and support on-premise deployment as well. This reduce the efforts to implement two systems.

**Ref:** Molehill provides complete support and optimization for cloud storage (including SCM).

### C++ Native Implementation

**Error Handling**

We need the ability to rewind from somewhere deep in the call stack, so it shall follow the try/catch C++ exception handling:

// Installed signal handlers for non-catchable errors  
SIGSEGV\_handler(…){err = PANIC, “internal error”};  
SIGTIMER\_handler(…){err = ERROR, “query timeout”};  
  
// Main process to process a query command   
Transaction \*tran = CreateTransaction();  
try {

PhysicNode \*plan = CreateQueryPlan();  
 plan->Exec([&](Row \*r){  
 …  
 });   
 tran->Commit();  
}   
catch (const std::logic\_error& e) {  
 tran->Abort();  
 …   
}

We also differentiate non-catchable errors and catchable errors. Here are the details:

* **Non catchable errors**: These errors, including SIGSEGV, SIGILL, SIGFPE, are unexpected and caused by internal bugs. We return FATAL in this case. We install a set of synchronous signal handlers to mark these errors only as it is not safe to directly handle it in signal handler, so check\_and\_throw\_on\_err () is invoked actively in unbounded loop to further throw it.
* **Try/catch-able errors**: These are explicitly thrown by engine code or the libraries it uses, like STL. We only throw 3 types of errors:
  + INFO/WARNING: We can simply log it.
  + ERROR: this means data contains error and query can’t continue. For example, for SQL math expression, we won’t rely on SIGFPE to handle div/0, instead we handle them by explicitly check the operands or results, then directly throw this error code.
  + FATAL: some internals error but disconnect can handle, say user password verification.
  + PANIC: this means some internal issue, which engine can’t handle due to logic inconsistency or failed too many times, so engine resort to restart.

**Memory Management**

Memory management considerations:

* Avoid wasting more memory than needed: In theory, the real memory usage needs to match the theoretical memory usage as much as possible.
* Avoid redundant new/delete operations: Consider output-scan query, scan can new one record at a time, and then output is responsible for deleting or notifying scan delete, but the record memory can be reused to avoid this new.
* Avoid redundant copy operations: Consider building tables on the build side of hash join, you can directly reference records without copying.

One option is to use C++ PMR for memory management, which is like PostgreSQL’s MemoryContext. The difference is, C++ PMR adds built-in support for containers, and we can use std::pmr namespace to access these containers instead of inventing our own containers. Another option is to use smart pointers like unique\_ptr, shared\_ptr: unique\_ptr is almost no overhead managing memory by implementing RAII semantics but shared\_ptr uses atomic instructions for thread-safety.

To management memory efficiently, we will overload new/delete, so PMR imposed virtual function call is unavoidable anyway. Memory allocator requirements:

* Life cycle
  + Temporary, operation (unit group, Expr Eval), statement, sub-transaction, transaction, session, server. Released at the end of the life cycle. So, there is no need to deal with one by one.
  + Even if the life cycle is used as a whole, deallocate needs to be supported, mainly to reduce unnecessary memory waste and the cost of reallocation: for example, if a tuple is reserved by the above operator (such as join build), then it will be longer, otherwise Just shorter.
  + Life cycle may change: For example, plan cache: Usually a plan uses a statement-level memory pool, and the memory is released after the statement ends, but if it needs to be cached, can it be saved without copying the memory pool used by the plan? In this way, the memory pool needs to be structured, and the sub-pool used by the plan can be removed so that it can not be deleted.
* Concurrency support
  + No concurrency: thread used by itself (including coroutine)
* Assign object characteristics
  + fixed size? Small?
* Profiling support
  + Used to diagnose leaks
  + Used to know the current memory usage

**Ref**: PostgreSQL MemoryContext

**Cluster Management**

Cluster management is required for shared nothing deployment.

# Interface

## SQL Interface

To implement a SQL parser:

* Use Bison instead of Antlr as the parser compiler
* Use an internal representation of parse tree, all parsers normalize to this format

There are at least two choices: Antlr or bison. Antlr provides a visitor model: whenever a rule is hit, you get a chance to go through it and generate your object.

For example, a function invocation is defined by the following rule:

function\_name '(' (expr ( ',' expr )\* )? ')' #FuncExpr

Thus the code to handle it is like this:

public override object VisitFuncExpr(FuncExprContext context) {

List<Expr> args = new List<Expr>();

foreach (var v in context.expr())

args.Add(Visit(v) as Expr);

return FuncExpr.BuildFuncExpr(context.function\_name().GetText(), args);

}

We will go with bison for performance considerations. My tests indicate bison is multiple times faster the Antlr with C++ target.

We can also further extend the SQL to make it more usable:

* Richer JSON support with “a.b.c” JSON path access. An alternative is supporting non 1NF table structure, say structure as a column. This is popular in NoSQL and google SQL family.

## APIs and Language Bindings

Apache Spark is leading in terms of interface flexibility. On the one hand, the DataSet API it provides makes SQL/MapReduce API, on the other hand, it integrates deeply with the language and provides better programming capabilities. The Spark API has basically become the de facto standard. Therefore, in addition to supporting the SQL standard, a new generation of databases should be able to adapt to the analysis interface like DataFrame.

APIs extension allow the application to reuse its parallel processing and analytics power. QPModel demonstrated a DataSet like interface which demonstrates language integration. Another example calculating Pi with Monte-Carlo can be found in Unittest/TestDataSet. A more complicated example can be found in section 6.1.

// leverage c#'s sqrt so no reinventing wheels

double sqroot(double d) => Math.Sqrt(d);

SQLContext.Register<double, string>("sqroot", sqroot);

// use sqroot in SQL

var sql = "SELECT a1, sqroot(b1\*a1+2) from a join b on b2=a2 where a1>1";

// above query in DataSet form

var a = sqlContext.Read("a");

var b = sqlContext.Read("b");

var rows = a.filter("a1>1").join(b, "b2=a2").select("a1",   
 "sqroot(b1\*a1+2)").show();

To support DataSet like APIs: (1) the APIs shall be translated into a canonical form that optimizer can recognize; (2) generate high level language bindings (like JavaScript, Python) so that users can easily reuse existing abilities.

### Multi-leveled APIs

It is possible to design the multi-leveled APIs to make it easy to be integrated with host database.

# Optimizer

QPModel pick up C# as implementation language because optimizer is logic centric, so a high-level language is preferred. After experiments, production may want to turn it into some C/C++ code, so the language must be a close relative of them. C# (.net core) provides some great features like cross-platform, LINQ, dynamic types to make modeling easy, and it is close enough to C++.

QPModel C# optimizer exercises the following constructs:

* Top down/bottom up structure: the optimizer does utilize a top down cascades style optimizer structure but optionally you can choose to use bottom up join order resolver. It currently uses DPccp ("Analysis of Two Existing and One New Dynamic Programming Algorithm") by G. Moerkotte, et al. It also implements some other join order resolver like DPBushy (TDBasic, GOO), mainly for the purpose of correctness verification. A more generic join resolver DPHyper (Dynamic Programming Strikes Back) is in preparation.
* Subquery decorrelation: it follows the "Unnesting Arbitrary Queries" and "The Complete Story of Joins (in Hyper)" by T. Neumann et al.
* CTE inline/non-inline (ongoing): it follows the Optimization of Common Table Expressions in MPP Database Systems by A. El-Helw et al.
* Cardinality estimation, costing: currently this follows "as-is" textbook implementation. We did not spend much time here because it is a local issue, meaning later improvements shall not impact the architecture. CE also demonstrate upgrade or version management.
* Distributed plan: follows the remote exchange model (gather, redistributed, etc.). A naive in-machine parallelism can be modeled with the same scheme.
* The optimizer also exposes a DataSet like interface which demonstrates language integration.
* Verify the optimizer by some unit tests and TPCH/DS. All TPCH queries are runnable. TPCDS we do not support window function and rolling groups. You can find TPCH plan here.

The optimizer takes a parsed plan includes the following major stages:

* semantic analysis and binding
* Substitution optimization: un-nesting, predicate pushdown, column elimination
* Cost based optimization: join-reordering, CTEs

Modern optimizers are already cost-based, and the new generation needs to be enhanced in several ways:

* Speed ​​of generating plan: Need to quickly generate execution plan
* The quality of the generated plan: The processing of CTE, SubQuery and Aggregation needs to be more comprehensive. Traditionally, the optimizer implementation mainly considers the size of the input, but does not consider the structure enough. Newer studies have shown that the shape of the execution plan (such as Acyclic Query) has a fundamental impact on query optimization.
* Learn the feedback from the actuator to optimize the next plan

We observed:

* Top down has the possibility of early-pruning, while Bottom-up has almost no
* Bottom up has a known fast connection optimization algorithm, while top down uses regular transformations, resulting in low efficiency
* Top down processing is not compatible with query rewriting (CTE inline/un-inline, eager/lazy aggregation, etc.) complex processing

Current idea is to generate multiple incompatible query plans through query rewriting, and each rewrite uses the Cascades method Top-down to start the optimization and calls the Bottom-up high-speed connection optimization core. Finally pick the best plan from it.

Because of multi-mode support, the optimizer, as the core of SQL, needs to be extended to support non-relational models: in particular, it needs to support stream data CQL and graph data PGQL. In terms of implementation, we need to give priority to supporting the relational model.

**Ref**: QPModel c#.

## Pre-Optimization

This includes binding, normalization and simplify query.

* Binding is the process to resolve each name in the query to the database or query object references.
* Normalization is to rewrite the expression into normalized canonical forms, so the later comparisons don’t have to deal with equivalents variants.
* Simplification stage is also called algebra optimization in textbook. The most important one is predicate push down, which push down the query predicate into the scan or closer to the scan to reduce rows flow in the plan. The former case is called SARG (search argument)-able scan. It can also include group-by columns reduction, generate implied predicates, remove redundant joins given PK/FK relationship etc.

### Overview

Binding starts with top level statement calling Bind() method. Classes derived from SQLStatement implement Bind()method as required for that class. What follows is mostly from the point of view of SelectStmt class (The class implementing Select statement) with some overview of other classes. All these classes are in the namespace of qpmodel.logic.

Select statement’s SelectStmt.Bind() is called with a parent context. Parent context will be null for the top-level statement (Select statement and others). It will be set to the parent statement or context when there is an outer level statement.

Select statement’s Bind()creates a new BindContext to signify the creation of a new context or scope for all the object references in this context or scope. This new scope links to the parent context if it was not null. BindContext includes a global subquery counter a Dictionary of all the tables references in the FROM clause of the current SELECT statement (or other statements as required), the current SQLStatement and the parent’s BindContext. This context is passed down as required.

SelectStmt.Bind() is called from Index creation (CreateIndexStmt), Subquery binding, binding of DML statements and binding of UNION SELECTS. UNION in this section refers to UNION, EXCEPT, INTERSECT with and without ALL/DISTINCT qualifier for brevity, these are all managed or encapsulated by the class called SetOpTree.

SelectStmt.Bind() does one of two things depending the statement being a part of UNION SELECT or not. If it is part of UNION it calls Bind() defined in SetOpTree to handle the binding of select statements in the UNION followed by binding the ORDER BY defined after the last select in the UNION, if there is one. SetOpTree.ind()will eventually call BindWithContext() in SelectStmt.

If the SELECT is not part of a UNION, then SelectStmt.BindWithContext() is called. This is the main driver of the binding process of a select statement. First, the FROM clause is bound so that all other column references can be validated and bound correctly. During the process of binding of table references in the FROM clause, BindWithContext()may be entered recursively to bind the subqueries and QyeryRef, FromQueryRef (also known as derived tables, query expressions etc.,) found in the FROM clause.

After the tables references and derived tables are all bound, the select list elements are bound one after another. This involves finding the table reference to which a column reference belongs to and setting the type information of these columns and the expression they are part of. After the select list element is bound it is normalized.

Next, the WHERE clause is bound and normalized if there is WHERE clause. Next, GROUP BY, HAVING and ORDER BY clauses are bound and normalized if they are present.

### Variable Binding

**Binding Table References**

A table reference in the FROM clause can be a simple base table reference (BaseTableRef), or an ExternalTableRef, or a table defined by one of QueryRef, FromQueryRef, JoinQueryRef (derived table). A base table is looked in the system catalog (Catalog class). If it is not found in the catalog, it is looked up the list of WITH clauses (CteExpr class) this FROM is part of.

If the named table is not found after all these searches, then an error is raised which ends the binding. If the named table is found it is checked for duplicated use (no two table refences should refer to the same table by name or alias). If the table is found and it is valid use, then it is entered into the list of table references found in the current FROM clause.

**Binding Select list expressions**

Although the section header mentions binding of select list expressions, the process of binding expressions in other contexts is almost the same. A select list element can be a \* (star), table.\*, simple column reference , a literal value, an expression or a scalar subquery returning single row and a single column. A \*, or table.\* is expanded to include all columns of all tables in the FROM clause and bound to the table to which each column belongs to.

All other elements in the select list are bound by the virtual method BoundAndNormalize in the Expr class. Some classes derived from Expr class override the Bind() method to customize the process of binding specific it. BindAndNormalize() calls Expr.Bind() followed by Expr.Normalize() to do the actual binding and normalization of the expression. Expr.Bind() calls Bind() method on each child first and replaces the original expression by the newly bound expression and resets aggregate table references, if any, in that expression to collect all tables referenced by the current expression, this list is maintained by tableRefs\_ in the Expr class.

Column references are bound by ColExpr.Bind() method. Column references bound by looking up the column reference by alias if one is provided or the name itself if there is no alias in the list of tables in the current context, if it is not found in the current context, it is looked up in the parent context recursively. If the named column is not found at the of this lookup, an error is raised. If the column reference is found in some parent context, then it is marked as a “Parameter” in this context and this context is now known to be correlated to the context in which the column reference was found and resolved.

A check is made to ensure no two column references are duplicates and that the a given column reference is resolved by one and only one table in the current context.

The ColExpr class represents a column reference or a expression with one column. The column reference’s ordinal is set to the ordinal position of the column in the table definition and the type to that of the type of the column definition in the table definition.

If the column is resolved to a FromQuery (derived table) it is DeQueryRef’d. This involves identifying the column’s underlying base table. This is done so that if remove\_from optimization is enabled there is no clash between the name of the column as it is known before the optimization and the name it will be known outside (inline version of the query) with the other names in the main statement.

Literals are represented by ConstExpr class, unary, binary and other function expressions are represented by UnaryExpr, BinExpr and FuncExpr classes respectively, most of them override Bind() and provide the specialization. For instance, BinExpr is specialization for all binary expressions. BinExpr.Bind() first lets the parent class do the binding of the two children. Next it does the semantic validation of the operation, types and other validation checks and sets up its own type based the type of the operation and the children.

When all children of an expression are bound the expression is marked as bounded.

**Binding WHERE and HAVING Clause**

The process of binding the WHERE clause is almost the same as binding any other expression with some extra processing. After binding the WHERE clause and normalizing it, it is possible that the entire WHERE clause may be replaced by a constant expression representing TRUE or FALSE indicating the fact that the WHERE clause evaluates unconditionally to TRUE or FALSE respectively. WHERE clause is validated so that there are no aggregate functions and that its type is Boolean. If remove\_from optimization is enabled, the WHERE and HAVING clause go through the process of DeQueryRef. The only difference in binding WHERE and HAVING is that HAVING can contain aggregates.

**Binding GROUP BY and ORDER BY Clause**

Before biding expressions in the GROUP BY and ORDER BY clause, the expressions are replaced by references to the respective expressions in the select list (order by 3, would make it order by the third select list element) if this is the case then the expression has already been bound. Other expressions in the GROUP BY and ORDER BY which are not position specifications are bound as any other expression. The presence of a GROUP BY expression sets a flag hasAgg\_ in the SelectStmt. Expressions in the GROUP BY are validated to contain no aggregates as arguments but if a FromQueryRef has been transformed (removed and merged or inline with the main query) then nested aggregates do not raise error. At the same time, if a FromQuery has been removed, each expression in the GROUP BY and ORDER BY goes through DeQueryRef.

**Binding Query Expressions**

In QPModel code, Query Expressions (and joined table) have many different variations and they are called QueryRef, FromQueryRef, CTEQueryRef, JoinQueryRef and they are represented by class named so. FromQueryRef class deals with queries appearing the FROM clause. Each of these queries represents a virtual or derived table. They may require having a name (derived table name) and name the output expressions (derived column names), known as outside[x].outputName\_. FromQueryRef maintains a map of the names in the derived column names and (known outside the query definition) and the expression/names they represent inside. The process of DeQueryRef helps resolve outside names to inside names and to the base table to which a given column belongs to. These are all bound like any other SELECT with a few differences.

**Binding Sub Query Expressions**

All kinds of subqueries are derived from SubqueryExpr class, which is derived from Expr class. They are bound through process like SELECT with additional validations such as a scalar subquery must return only one column, one row etc. The entry point for binding subqueries is SubqueryExpr.BindQuery(). A SELECT statement (top level query) doesn’t have a DataType but other kinds of subqueries have a type. Binding sets up this type.

### Normalization of Expressions

Expressions are bound and then they are converted to a normal or canonical form so that two semantically equivalent expressions written in different form/structure can be identified quickly. Another goal is to simplify the expressions as much as possible during the compile phase. Normlizer.cs implements Normalize(), a virtual method of Expr class.

Each derived class of Expr may override Expr.Normalize() specialize the needs of normalization of that class but only FuncExpr, CoalesceFunc, UnaryExpr, BinExpr, and CastExpr override to specialize normalization.

In order to avoid clutter in all the classes that must specialize normalization, Expr and other classes which must specialize Normalize(), they have been declared as partial classes and their Normalize() method is implemented in Normalize.cs.

Binding phase calls Normalize()on each expression after it has been bound. The base version simply calls Normalize()on all children of the current expression and returns possibly modified expression.

Logically, normalization does the following transformations to the expression. They are not implemented in the exact manner they are described to avoid traversing the expression tree

several times and more than one transformation could happen at once. In the rules below 'op' and 'comp' denote a generic arithmetic and comparison operators when specificity is not required. We have implemented the following normalizations:

**Constant move**: Bring all possible constants together so that later transformations can simplify or even remove some of the constants:

* CONST + expr => expr + CONST
* CONST \* expr => expr \* CONST
* CONST comp expr => expr ~comp CONST
* CONST < expr => expr > CONST
* CONST > expr => expr < CONST
* expr op CONST1 comp CONST2 => x comp CONST2 ~op CONST1
  + Ex. x + 1 comp 10 => x comp 10 – 1, and later it becomes 9

**Constant folding**: Replace expressions involving constants with the value of that part of the expression.

* CONST op NULL => NULL
* CONST op CONST => EVAL
* FUNC(CONST) => EVAL. FUNC is one of the aggregates MIN, MAX, AVG or other non-aggregate function.
* FUNC(NULL) => NULL. FUNC is one of the aggregates MIN, MAX, AVG, SUM or other non-aggregate function.

**Arithmetic Simplification**: Eliminate unneeded computations:

* expr op NULL => NULL: Except when op is IS or IS NOT, in which case it values to TRUE or FALSE.
* expr + 0 => expr
* expr - 0 => expr
* expr \* 1 => expr
* expr / 1 => expr
* Distribute multiplication over addition when there are constants. Ex. (x + 5) \* 10 => x \* 10 + 50
* Distribute multiplication over multiplication where there are constants. Ex. (x \* 5 ) \* 10 => x \* 50.

**Coalesce simplification**:

* COAL(x, a, b, .. const, …) => EVAL to const

**CAST simplification**. Eliminate CAST of CAST of CAST and so on when the argument is constant.

**NOT simplification**:

* NOT NOT X => X
* NOT (X AND Y) => NOT X OR NOT Y
* NOT (X OR Y) => NOT X ND NOT Y

**Comparison Simplification**. Eliminate unneeded comparisons:

* CONSTEXPR1 comp CONSTEXPR2 => EVAL

**Relational operator simplification**:

* X + CONST1 = CONST2 => X = CONST2 – CONST1
* X + CONST1 > CONST2 => X > CONST2 - CONST1
* X + CONST1 >= CONST2 => X >= CONST2 - CONST1
* X - CONST1 = CONST2 => X = CONST2 + CONST1
* X - CONST1 >= CONST2 => X >= CONST2 + CONST1

**Logical simplification**:

* CONST1 AND CONST2 => EVAL
* CONST1 OR CONST2 => EVAL

There are a few more simplification transformations possible but not implemented at this time.

## Subquery decorrelation

Subquery decorrelation is part of simplification stage.

## The Cascades Framework

There are several major steps implements the Cascades framework:

* Exploration to enumerate all promising alternatives
* Enforce property to further rich alternative set
* Associate costs with each alternative and pick up the minimal as output

### Exploration and Implementation

Exploration stage prepares possibilities for the implementation stage. After we get all logical trees, query optimizer enters implementation stage, where each logical operator is transformed into a set of physical operators. There is a central concept called *properties*, which list certain attributes for a PhysicNode, for example, ordering.

Only after implementation stage is done, we can calibrate cost and pick up the winner. The execution tree cost is a sum of all iterators. The iterator cost is sum of I/O cost and CPU cost. We can observe costs associated with each iterator in SQL Server via query plan.

One of the most important transformation in the exploration stage is the join ordering. As we mentioned earlier, it can easily generate huge number of combinations as involved tables increase. This problem gets agitated with database views and recursive queries: Say the WITH clause may involve 5 tables, and the query body uses the WITH clause table 3 times, then we will have 15 tables involved. We can look at a 3-tables inner join query R × S × T which has 12 logical equivalent join orders.



Figure 3‑1. 12 different join plans of 3 tables join in exploration stage. Each tree has 5 iterators; thus, we will use memory to hold 60 iterators to represent these variants. This does not include plans generated on physical implementation stage. If we consider each join have 3 physical implementations: nested loop joins, hash join and merge join. Then totally different physical plan trees are 12 ×3×3=108. Again, each tree has 5 iterators, thus we need a memory space holding 540 physical iterators to accommodate these variants.

There are problems arising now: first, how do we enumerate plans and without repeating themselves? Second, for many table joins, the number of possible plans will be huge, how can we manage the memory? To enumerate plans, we use *optimization rule*, which takes a fragment of a plan and outputs an equivalent fragment. In the exploration stage, the optimization rules transfer one to a logical equivalent. In the implementation stage in next step, it transfers to its physical form. To avoid repeating and efficient use memory, memorization is needed. In above 3 table join example, we can see that though there are 12 plans, some fragments show up several times, like R×S show up twice. So instead of recreating a totally new tree each time, we can use pointer to point to the existing one. Here we introduce a concept *optimization group* which gathers equivalent fragments together. Now starting from initial input plan R × S × T and load it bottom up, we get this:

GROUP 4: Join(2, 3)

GROUP 3: T

GROUP 2: Join(0,1)

GROUP 1: S

GROUP 0: R

Suppose there are two join rules used in the transformation:

RULE 1(commutative): R × S = S × R  
RULE 2(associative): (R × S) × T = R × (T × S)

With these rules, we can enumerate plans starting from the top. First with commutative rule, we get one new member in group 4:

GROUP 4: Join(2, 3), **Join(3, 2)**

GROUP 3: T

GROUP 2: Join(0,1)

GROUP 1: S

GROUP 0: R

Then apply associated rule to both members in group 4:

GROUP 5: **Join(1, 3)**

GROUP 4: Join(2, 3), Join(3, 2), **Join(0, 5)**

GROUP 3: T

GROUP 2: Join(0,1), Join (1, 0)

GROUP 1: S

GROUP 0: R

Now the top is group 5. Apply rules (only commutative rule works), we get

GROUP 5: Join(1, 3)**, Join(3, 1)**

Group 5 is done exploration. Group 4 becomes top again. Apply rules: commutative rule gives us

GROUP 4: Join(2, 3), Join(3, 2), Join(0, 5), **Join(5, 0)**

Associative rule gives us another new member in group 4 and a new group 6:

GROUP 6: **Join(0, 3)**  
GROUP 4: Join(2, 3), Join(3, 2), Join(0, 5), Join(5, 0),  
 **Join(1, 6)**

Now the top is group 6. Keep above procedure till all groups are optimized: We shall have the following sequence:

GROUP 6: Join(0, 3), **Join(3, 0)**

Next:

GROUP 4: Join(2, 3), Join(3, 2), Join(0, 5), Join(5, 0),  
 Join(1, 6), **Join(6, 1)**

Next

GROUP 2: Join(0, 1), **Join(1, 0)**

Then there is no more group to optimize, so finally we get Figure 3‑1 equivalent with the following compact form. Group 4 is all equivalents of R × S × T, group 2 is all equivalents of R × S, and so on:

GROUP 6: Join(0,3), Join(3,0)

GROUP 5: Join(1, 3), Join (3, 1)

GROUP 4: Join(2, 3), Join(3, 2), Join(0, 5), Join(5, 0),

Join(1, 6), Join(6,1)

GROUP 3: T

GROUP 2: Join(0,1), Join (1, 0)

GROUP 1: S

GROUP 0: R

In above procedure, we load the initial query tree in a bottom up order. We shall remember that group 4 is the root of the query tree. If we load the initial query tree top down, then group 0 is always the root. Here are the groups we get. This is equivalent to Figure 3‑1 as well:

GROUP 0: join (1, 2), join (2, 1), join (5, 4), join (4, 5),

join (6, 3), join (3, 6)

GROUP 1: join (3, 4), join (4, 3)

GROUP 2: T

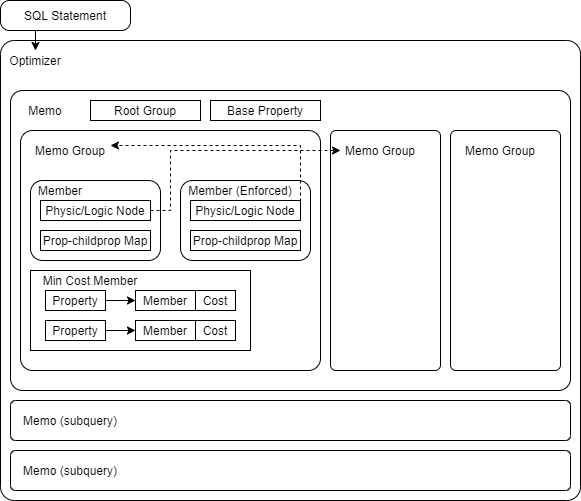
GROUP 3: R

GROUP 4: S

GROUP 5: join (2, 3), join (3, 2)

GROUP 6: join (2, 4), join (4, 2)

One thing to note is that with above changes, we reduce logical operators stored in memory down from 60 in Figure 3‑1 to 15.



This figure demonstrates the layout of all the relevant objects within the query optimizer. All the objects mentioned below are defined in optimizer.cs:

* Optimizer: Lies within each SQL Statement object.
* Memo: There could be multiple memos, which corresponds to each logic tree. A cached subquery may have its own memo object within the same optimizer.
* Memo group: A memo consist of multiple memo group, which corresponds to each node in the logic tree.
* Member: A memo group consists of multiple members, which corresponds to each logically equivalent LogicNode or PhysicNode within the same group. There are two categories of members: members generated during exploration and members added during property enforcement (see CGroupMember.isEnforcer\_).
* Property: The property requirement object.

### Property Enforcement

Property enforcement is to solve the order and distribution requirements of the query:

1. Order: the ORDER BY clause of the query, which consists of a list of sorting expressions and its descending/ascending requirements.
2. Distribution: the internal physical node requirements to handle data distributions, which applies to both within-machine data partitioning and cross-machine data distribution. There are three distribution types (see DistrType):
3. Singleton: data not distributed, all data are within one machine
4. Distributed: data Distributed on a specific list of expressions
5. Replicated: data is Replicated among all the machines.

Besides above, we also introduced an AnyDistributed type, which means data is Distributed, but the list of expressions is not specified.

Properties is propagated via a top-down manner. They are required on root of the memo groups, and requirements may be propagated to child groups or new requirements may be imposed by certain physical nodes (for example, PhysicStreamAgg requires sort order). By default, the required property of the final output is <Singleton, no order>. Before we start the property enforcement, logic nodes like LogicSort are transformed into requirement, thus we will have <Singleton, Order<…>> to begin with. {always <singleton, …>, even for subquery}

Likewise, finding the optimal plan is a top-down recursive calling process. The function call CMemoGroup.CalculateMinInclusiveCostMember(properties) uses current memo group and the property requirement properties on that group and returns the member with the minimum cost. For each function call, first the required property is transformed to a list of sub-properties (i.e., property relaxation) via GenerateRelaxedProperties() that can turn into the current requirement by forcing a single PhysicNode. For example, <Singleton, order(a1)> has sub-property <Singleton, no order> since it can be turned into <Singleton, order(a1)> by enforcing a single PhysicOrder(a1) node. Each sub-property corresponds to a new enforcement member being added to the member list. All enforcement nodes are single physic node with only one child—the PhysicMemoRef referring to the members within same group but with a different property requirement.

Each sub-property will be called upon to find the optimal member for that specific property. All the property requirements, including the relaxed sub-properties, and their corresponding optimal member is kept in a dictionary (see CGroupMember.minMember\_) within the memo group object. To avoid redundant calculation, if CalculateMinInclusiveCostMember() is called upon with the property requirement that has been called before, the recorded optimal member will be directly returned.

**Property Relaxation**

Property relaxation refers to the transformation of a property requirement to a list of less restricted property requirement. Then this less restricted property can be transformed back to the original property by enforcing some physic operation.

* Order: Order relaxed to no order:

<Singleton, order(a1,b1,b2)> -> <Singleton, no order>

Order must be Singleton; this is asserted throughout the process. Apparently, order can be enforced by introducing a PhysicSort. If a requirement is <Singleton, order(a1)>, the received data stream of <Singleton, order(a1,b1)> would not require a PhysicOrder node sorting the whole thing again.

* Distribution
* Singleton relaxed to AnyDistributed or Replicated:

<Singleton, no order> -> <AnyDistributed, no order>, <Replicated, no order>

Singleton can be achieved by AnyDistributed or Replicated through PhysicGather. For AnyDistributed, data is gathered over all the running machines; For Replicated, data is gathered from only one machine.

* Distributed relaxed to AnyDistributed:

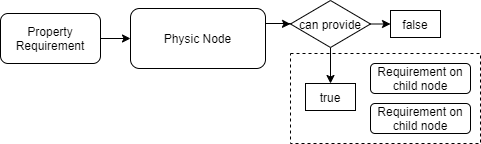
<Distributed(a1), no order> -> <AnyDistributed, no order>

This provides the possibility that we can change the distribution on the fly: distribution on a certain list of expressions can be achieve through redistribution, and the previous distribution does not matter if it is also Distributed.

For example, <Singleton, order(a1)> can be relaxed to <Singleton, no order> , then it further relaxed to {<AnyDistributed, no order>, <Replicated, no order>, <Singleton, no order>}.

**Property and physic node**

Property is always required upon the physic node instead of physic node providing a list of possible properties. It is designed this way because a node may be able to preserve certain property from the child group. For example, PhysicNLJoin/PhysicHashJoin can preserve the order of the outside node/probe side node, and the specific order depend on the outside child group, which cannot be known in advance. There is a method PhysicNode.CanProvide() for physic node object to determine if the it can provide the required property. For the PhysicNLJoin/PhysicHashJoin case, now the property requirement can be propagated to build end child group. If the physic node can provide that property, the property requirement on its children is also returned.



The figure above demonstrates how the property interacts with physic node. Below is a non-exhaustive list of notable physic nodes:

* PhysicScanTable: The distribution method (see TableDef.DistrMethod) of table can be one of NonDistributed, Distributed, Replicated, and Roundrobin. Singleton and Replicated property are directly matched with NonDistributed and Replicated. If the list of expression for distribution is an exact match with the distribution column, then it is considered supplied. If the requirement is AnyDistributed, then either of Distributed or Roundrobin is acceptable.
* PhysicHashJoin: Hash join is one of the most complicated physic nodes due to its versatility with distribution requirement. It can provide Singleton and Replicated, and the requirement on children is also Singleton and Replicated respectively.

It can also provide Distributed, if the join key contains the expression list and they are matching. The requirement on children could be any of the following:

* + (build side / left key: Distributed, probe side / right keys: Distributed)
  + (Replicated, Distributed)
  + (Distributed, Replicated)

It can also provide AnyDistributed, the constrain of join key match is lifted.

* PhysicStreamAgg: Can provide the order on the aggregated expressions, and the requirement on the child is also aggregated expressions must be sorted.

**Query Example**

Given table a as listed below, and table b having the same data.

|  |  |  |  |
| --- | --- | --- | --- |
| a1 | a2 | a3 | a4 |
| 0 | 1 | 2 | 3 |
| 1 | 2 | 3 | 4 |
| 2 | 3 | 4 | 5 |

Take this query as an example: SELECT a1, b1 FROM a, b WHERE a2>b1 AND a1>=1 ORDER BY a2. The logic tree of this query is:

LogicOrder

Order by: a2[1]

-> LogicJoin

Filter: a2[1]>b1[0]

-> LogicScanTable a

Filter: a1[0]>=1

-> LogicScanTable b

Without property enforcement, this query will just be transformed into physic tree of very similar layout, with PhysicOrder on top. The transformation will be limited to the Join(a, b) or Join(b, a). However, it is not the optimal plan if join is many to many as PhysicOrder having more rows to sort.

The physic plan generate with property enforcement is shown below: Sort is first remove and being treated as a Sort requirement, thus finally PhysicOrder is pushed below as a property requirement imposed on the Memo, and PhysicNLJoin is able to propagate the order requirement downward.

PhysicNLJoin (inccost=163.58, cost=156, rows=3) (actual rows=5)

Output: a1[0],b1[2]

Filter: a2[1]>b1[2]

-> PhysicOrder (inccost=4.58, cost=1.58, rows=2, memory=16) (actual rows=2)

Output: a1[0],a2[1]

Order by: a2[1]

-> PhysicScanTable a (inccost=3, cost=3, rows=2) (actual rows=2)

Output: a1[0],a2[1]

Filter: a1[0]>=1

-> PhysicScanTable b (inccost=3, cost=3, rows=3) (actual rows=3, loops=2)

Output: b1[0]

This optimization is also the case for distributed plan. Without property enforcement framework, we may only generate naïve plans with tables are directly redistributed before the hash join to match the partition alignment requirements, and that method may output suboptimal physic plan for queries involving a mix of distributed and replicated tables. Now the property requirement may be pushed down to do in advance or delayed to the top depending on the cardinality to test costing of all possible shuffling plans.

### Find Min Cost Member

Finding min cost member is the last step of cascades framework (CMemoGroup.CalculateMinInclusiveCostMember).

After relaxation of less restrictive sub-properties, we look for the member with minimum cost. This consists of two parts: go through member list to find directly supplied members and go through sub-property list to find enforced member by enforcing a physical node (cost).

For each member, if it is a physic node, it will be checked if the required property can be fulfilled. If so, the total cost of that member will be calculated by summing the cost of that node plus the minimum costs of child group members that can provide the property requirements imposed on the child groups. The “can provide” specifics of this step are discussed in the property and physic node paragraph.

Similarly, for each sub-property, since the finding minimum function is already called upon, the total cost would be sum of enforcement node cost and the total cost of the member fulfilling the sub-property. When the recursive calls are complete, the min-cost member will be extracted to form a tree of physic nodes without PhysicMemoRef, and this three will be further used for execution.

## Other Topics

### Staged Optimization for OLTP Queries

SQL Server follows the Cascade optimizer. SQL Server has more than 300 optimization rules, which can transform query tree into many equivalents. If we apply all these rules in one shoot, the timing spent on optimization could be too long. This is especially bad for OLTP queries, which have short execution time and some simple transformation shall generate optimal plans. So, SQL Server cut the optimization into several stages, and each stage contains a subset of rules. At the end of each optimization stage, compiler decides if it shall proceed to next stage or the cost is low enough.

In general, database implementers do not consider query optimization parallelism for several reasons: One is that the query optimization time in OLTP is trivial thus no worry. For OLAP complex many joins queries, the portion of the optimization may take seconds, but this is again trivial compared to the query execution time. In many situations, database engine can cache the plan thus next time the query optimization won’t take long. Another reason is that query optimizer is quite complicated with a lot of heuristics hardcoded, which may imply orders, thus it is not an easy work to make it parallel. For the Cascades query optimizer framework, we have seen above, it is much easier to make it parallel.

### Integrate with ML based methods

Advanced statistics will affect the calculation of the cost model. This is a partial change and does not affect the existing optimizer structure. The ML algorithm will not affect the transformation and implementation rules based on logical equivalence, because this is determined by relational algebra; however, it may affect the search and pruning algorithms based on MEMO-the current judgment is: it will not lead to the above optimizer Refactor.

On some fundamental subjects like cardinality estimation, recent research indicates ML based methods can beat classic methods by a large margin. Engineering wise, it is possible to integrate ML based methods like CE with optimizer in decently as the CE interface are clean.

However, to get these methods ready, there are still some research needed:

* Preparation time: classic CE needs a sampling or full scan to collect stats while ML needs training and training in general takes way longer time and GPU/DPU requirements.
* Inference time: some methods (Naru) needs runtime sampling which could be unacceptable cost for OLTP low latency queries.
* Non-linear and hard to explain results: this is fundamental and well known. For example, “a>100” may yield speculated smaller result set size than “a>10000”.

### Other Items

**Parameter query optimization**: PREPARE Query with parameters are important for performance but they are hard issues for optimizers.

**Optimizer versioning**: QPModel C# has demonstrated cardinality estimator versioning (see CardEstimator). There are other related technologies like FORCEPLAN.

**Adaptive query processing support:** This is discussed in detail in section 4.4.1 from executor point of view. Optimizer design can follow the framework there to provide support from optimizer’s context.

**Peephole executor optimizations**: There are execution decisions may affect performance but not falling into fully fledged optimization frameworks we introduced in previous sections, which we call peephole executor optimizations. They may include (1) column trimming; (2) row buffer management (eager/lazy). We will talk about them at different places.

# Executor

Here are key technologies used include:

* Row/Vectorized operators for instruction reduction and better IPC. Ideally, Intel processors boast IPC around 4 and we will target 2.
* LLVM runtime code generation to further reduce instructions. Booster core shall match hand-writing code with an average extra 10% instruction overhead.
* AVX2 enhancement for better instruction efficiency. We will leverage it 256bit operands width.
* Resource tracking and adjustment with compliance to resource budget
* Adaptive strategy to reduce reliance on optimizer’s decisions. This is already partially done within Spark after shuffle stage. We may consider more within Booster.
* Fine granular resilience by introduce some materialized node into the plan (patented).

As Molehill already demonstrated, no matter what form of DataSet API applications: direct API calls, SQL or structural streaming, they will finally transform into its physical plan form and Engine will execute this physical tree natively. Visually, we can explain() to verify the plan. Here are some examples:

* **DataSet API:** Here is an example of Spark-like DataSet API application that Engine can handle:

Dataset<Row> people = session.read().parquet("...");  
Dataset<Row> department = session.read().parquet("...");

people.filter("age".gt(30))  
 .join(department, people.col("deptId").equalTo(department("id")))  
 .groupBy(department.col("name"), "gender")  
 .agg(avg(people.col("salary")), max(people.col("age")));

They are doable because no JVM involved. Notably, we won’t support the following important operators because of JVM functions involved: (1) map: user can use SELECT projection to partially overcome this. (2) flatmap: Engine can flatten the results but can’t apply the map function. (3) groupByKey: user can use GROUP BY to partially overcome this.

* **Structural Streaming:** Here is an example of supportable structural streaming by introducing window operator:

Dataset<Row> words = ... // streaming DataFrame

// Group the data by window and word and compute the count of each group  
Dataset<Row> windowedCounts = words  
 .withWatermark("timestamp", "10 minutes")  
 .groupBy(  
 functions.window(words.col("timestamp"), "10 minutes", "5 minutes"),  
 words.col("word"))  
 .count();

* **SQL:** Above DataSet application essential equals to a SQL statement:

SELECT MAX(age), AVG(salary) FROM people   
 JOIN department ON deptId = id   
 GROUP BY department.name, gender;

So it is executable in Engine. But if UDF is involved in SQL, we can’t fully handle them:

def makeDT(date: String, time: String, tz: String) = s"$date $time $tz"  
sqlContext.udf.register("makeDt", makeDT(\_:String,\_:String,\_:String));

sqlContext.sql("SELECT amount, makeDt(date, time, tz) from df");

In the executor core, it shall support streaming and traditional batch shall be treated as a bounded form of the streaming. APIs expose the full power of the engine than SQL. Overall, executor shall follow a multi-engine design to support multi-modal scenarios.

## Overview

If we look at the most recent IBM DB2 benchmark on 02/22/2013 (they stopped the game afterwards). TPMC value is 1,320,082 transactions per minute. Since TPMC is counted on NewOrder transaction, which is around 45% of the total transaction, thus it is about 2.9 million transactions per minute on the system, which is about 48.9K transactions per second. Notice the system has 16 Intel Westmere processors at 2.9G (not counting HT), assuming CPI=2, thus the system can retire 16\*2.9/2 = 23G instructions per second. This gives us 23G/48.9K=470K instructions per transaction. Some recent TPCC benchmarks scale out well with humongous TPMC number but per transaction performance is worse due to scale out overhead. So a target with 10X improvements on the small to medium scale OLTP performance, which is challenging but possible. On the other hand, TPCH benchmarks can get up to 1\*CPU\*Second/GB for power run, which is quite close to the performance limit, so our target shall be matching these performance numbers.

Benchmarks can fix query plan relatively easy, so executor and storage engine are the main source for the benchmark performance. We will focus on the executor on this section. The executor will follow the callback implementation as demonstrated in QPModel C#. Here is an exemplar code from PhysicStreamAgg implementation:

// aggregation is working on aggCore targets

child\_().Exec(l => {

if (context.option\_.optimize\_.use\_codegen\_)

// codeGen code …

else

{

var keys = KeyList.ComputeKeys(context, logic.groupby\_, l);

if (curGroupKey != null && keys.Equals(curGroupKey))

// new group code …

else

// finalize current group, callback and start a new group  
 callback(r);  
 }

});

There two parts of this code:

* The regular non-codeGen part. The focus here is to illustrate how the executor uses the physic plan generate by the optimizer to evaluate the query tree and expression.
* The codeGen part follows the similar code as the non-codeGen part. This is important to make the codeGen easy.

What is not covered in C# code with native implementation details will be covered here.

**PhysicNode** **Hierarchy**

Class PhysicNode is the parent class of all physic node implementation, say PhysicHashJoin, PhysicAggregate, PhysicScan etc. PhysicNode is not only limited to relational algebra functions, it shall also extensible to include other functions like profiling, sequencing, branching or loops. In this way, an execution tree can represent any programs.

### Resource Management

Resource management taking care of the following things:

* Estimate resource budget (especially memory) and report back to engine for cluster RM’s requests.
* Negotiate resource budget during query execution when actual is off the estimation.
* Work with other components, comply with the contract with other execution engine to complete query execution within budget.

We must start with current phase to do RM.

**Resource estimation** All physical operators shall implement an EstimateMemory() interface to return its own resource requirements. The catch here is that some early resource release is possible, so this need to be handled.

**Negotiate resource** Because the actual data amount might be quite off the estimation due to planner’s restriction. We shall give executor a chance to negotiate this. The negotiation could success or fail. Upon failure, RM shall keep comply with its previous agreement with optimizer.

**Comply with budget** This is the contract between Optimizer and Executor. RM provides the resource grant mechanism to tracking them and all major engine resource usage (like hash table, sorting pool etc but excluding meta-data) is throttled by it.

### Profiling Framework

There are several profiling methods we usually support:

* EXPLAN command: To support runtime explanation, we will introduce a separate Profiling node instead of kludge profiling code inside each operator. **Ref**: QPModel
* Counter framework: There shall be levels of query-able counters to reflecting internal status, which shall include all important database external and internal controls and objects. Meanwhile, we can also introduce internal resource counters for CPU and memory for fine granular profiling besides what OS provides. **Ref**: SQL Server DMV.

Like we design programs for better performance, scalability and supportability, we shall design the program better for profiling. Here are some guidelines:

* Profiling has overhead. More profiling data collected, bigger the overhead.
* Profiling from multiple aspects and correlate them.
* It is important to mapping profiling data to program objects/components.
* It is useful to keep a record of historical profiling data.

PostgreSQL does profile like this in its major pipeline loop:

TupleTableSlot \*ExecProcNode(PlanState \*node){

if (node->instrument)

InstrStartNode(node->instrument);

switch (nodeTag(node)){

case T\_SeqScanState:

result = ExecSeqScan((SeqScanState \*) node);

...

}

if (node->instrument)

InstrStopNode(node->instrument, TupIsNull(result) ? 0.0 : 1.0);

return result;

}

For each row output from each iterator in the pipeline, when profiling is not enabled, there could be two extra memory loads and two possible branch-misses, which is bad for light weight iterators. When profiling is enabled, PostgreSQL pays two function calls with a timestamp retrieval call within each. In spirit of engineering excellence, can we do better than this? There could be many alternatives:

* We can use hot patch based dynamic detouring method to inject instrumentation code when profiling is enabled. In this way, there is almost no impact on performance when profiling is disabled.
* We can create a new physic node PhysicProfile, which is simply a wrapper on its child iterator. When query profiling is enabled, we will add this iterator before each original iterator, thus this iterator can work like a water meter in the pipeline. This method also has the merit that it does not impact performance when profiling is disabled. Also, this method can well separate the profiling logic from iterator implementation. The problem is that it has deeper pipeline depth at profiling, thus maybe slower than current method.
* To reduce the per-row overhead at profiling, we could do sampling: profiling one row for every 16 rows. This essentially not different from current method: To capture timing of processing a single row, we need timer with precision of 100s of cycles. But current timer (gettimeofday in UNIX) does not provide such precision, so current method is in fact sampling on time. This sampling shall work very well if we have enough rows flow in the pipeline, but this shall satisfy well our purpose of profiling to identify the expensive part in the query.

Vectorization (see section 4.2) significantly to reduce per-row overhead at profiling is to batch rows, thus overhead is amortized among these hundreds of rows. This method not only reduces the profiling cost but also highly improved pipeline efficiency.

### Expression Evaluation

The expression evaluation done in QPModel C# is following classic pull method. For example, a binary operation (l, +, r).eval() = l.eval() + r.eval(). This is good for codeGen purpose as codeGen will essentially generate similar efficient code. This is however not efficient because of per value virtual function call and value memory manipulation overhead. A more efficient way for expression evaluation is stack based where we separate expression evaluation into two stages: initialization per query and evaluation per row.

The initialization has two steps: tree enqueue and slot allocation. The tree enqueue decides the FIFO order which expr get evaluated first. This follows post-order per correctness. This order also decides how many slots we need for evaluation. Look the example below, if we switch the left right tree, the slot tree can become 0(0(0)(1))(1) which means we only need 2 slots instead of 3. Slot allocation decides where the expr dump the results. Basic rule is if current node is parent's nth child, it will use slot parent + nth. The expression thus can reuse slot for temp results dumping without dynamic memory allocation for every expression. The evaluation procedure loop through the generated queue, evaluate each expression and write to the specified slots. Here is an example:

0 4

/ \ / \

0 1 0 3

/\ / \

1 2 1 2

[slot] [enque]

### Basic Execution

**Coroutine Based** C++20 introduced coroutine as the standard way to write async program. A coroutine represents a function. A group of coroutines is carried by a single thread and can be switched between coroutines at a cost of < 100ns. This provides a more efficient way than traditional multi-thread switching for CPU reuse. C++20 incorporated coroutine into the standard in May 2019, which also provides implementation possibilities for coroutine-based executor.

**Row Management** Row management manages the memory used for row passing. In general, row passing follows the ownership method like unique\_ptr: the callee pass the row up to the caller and the caller return true/false indicate if the callee can reuse the buffer. Once the caller tells true, the caller shall assume the row will change immediately. Otherwise, the callee shall be responsible not altering the row and the row memory ownership becomes the callers.

There could be tradeoffs for multiple alternatives of managing row buffers (say eager copy or lazy copy on scan side), we shall let optimizer to join the decision process to pick up which one is the best for performance.

### CodeGen

The same code of PhysicStreamAgg in section 4.2 but let’s look at the codeGen this part:

// aggregation is working on aggCore targets

child\_().Exec(l => {

if (context.option\_.optimize\_.use\_codegen\_){

var lrow = $"r{child\_().\_}";

context.code\_ += $@"

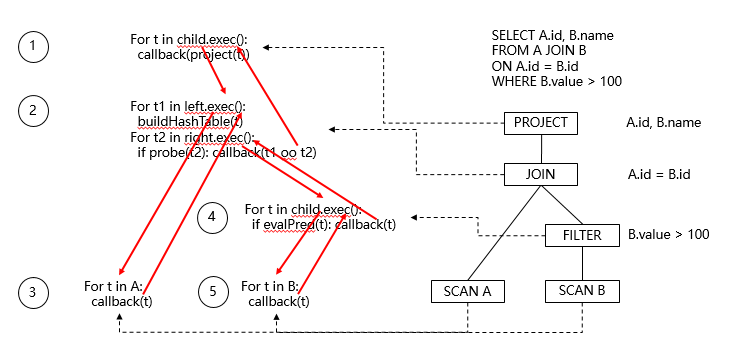
var keys = KeyList.ComputeKeys(context, {\_logic\_}.groupby\_, {lrow});";

context.code\_ += $@"

if (curGroupKey{\_} != null && keys.Equals(curGroupKey{\_})) …

Above procedure is quite like the interpret implementation: it follows the same logic flow and only replace the compiling time decidable variables, say current logic node, aggrecore.Count etc.

Below is the illustration of a query how a query is executed with codeGen generating code:



This method shall also be extendable to language integrated store procedures.

**Ref**: QPModel codeGen implementation.

### Streaming Support

## Vectorization

Like CodeGen, vectorization can also reduce redundant branch judgment and call prologue/epilogue, providing several times the performance improvement. On the other hand, vectorization is still an interpreted execution mode, and no compilation process can be executed immediately. Vectorization also provides a more effective data structure for AP big data processing methods, making SIMD optimization possible. The trend is that both AMD64 and AARCH64 continue to enhance SIMD. On this basis, some basic algorithms such as Join have some optimization algorithms in recent years, such as CHT, TrieJoin, etc., all need to be followed up.

Vectorized query execution is a significant performance improvement over the current row pipeline execution engines, which are used by most traditional database engines. In the traditional engine, the data unit passed between iterators is a row, while the vectorized query execution uses vectors. Using vectors as a data unit has several benefits:

* amortize the per-row overhead to a vector of rows
* CPU cache and instruction pipeline friendly
* natural fit for columnar storage
* opportunities for SIMD acceleration

The optimizer is responsible for putting v-Iterators and existing iterators together to make a performing plan for a query. As the column store can provide vectors naturally, so we can directly put v-Iterators on top of them. As the vectorized iterators have different interface from the traditional row iterators, we must convert the data format back and forth to enable a correct implementation. About the cost of new v-Iterators introduced, we can set the cost of it as 1/10 of its row-based iterators. By tuning the cost of V2R, R2V iterators, we can make certain vectorization on row store plan possible or impossible.

The core data structure of vectorization is CBatch, which shall contain the following essential information:

* data set meta information, including row format.
* Informational flags per row. This may include selections etc.
* Column vectors: we will have different sized vectors for different data types.
* Informational flags per value per column. This may include NULL bits or data type specific flags.

A selection basically means if the corresponding row shall be included in the processing or not. There are two major choices for representation: one is bit based: if 1 means selected and 0 means not. The other is position based, so a selection is a uint16 integer, which indicates the row offset of the selection. OpenGauss uses the former method and we will use the latter method.

With codeGen’s help, it is possible to omit certain fields if DDLs give a clear hint: for example, we won’t generate NULL bit checks related calls for non-nullable columns. We omit these details now and sketch the major part of CBatch like this:

class CBatch {

// each value is a position to a column value in ScalarVector  
 uint16 selection\_[cRows\_];

// list of normalized columns  
 ScalarVector columns\_[cColumns\_];  
};

Bigger CBatch size (cRows\_) yields to better the performance in terms of amortized overhead. However, bigger size also means bigger chance of CPU cache trashes, so depending on the plan shape and memory requirements of each iterator, a balance point is possible. We fix MaxBatchSize to 1000 to avoid repeatedly memory allocate/free process. It is also clear that optimizer column trimming may be optional for row stores but for sure needed for columnar storage.

A ScalarVector is an array of scalar values and a scalar value is a datum with size 1 to 8 bytes. It shall further enhance with type information, const (RLE) / nullable etc.

template<class T>  
class ScalarVectorT: ScalarVector {

// list of scalar values, varying size between 1 ~ 8 bytes  
 T values\_[cRows\_];  
 };

For column vector, an alternative is to normalize all data types to equal width data types, say uint64. This has the benefits for ease of coding without introducing switch/case within per value per column loops, but with LLVM based code generation, we don’t need the branches, so this is not a problem anymore. Meanwhile, this choice is especially good to leverage SIMD.

The above changes can be best illustrated by a typical aggregation query plan:

PhysicSort (L\_ReturnFlag,L\_LineStatus)   
 PhysicHashAgg (L\_ReturnFlag, L\_LineStatus)  
 PhysicScanTable (LineItem + filter)

When the vectorized hash aggregation is implemented, the optimizer can generate this plan:

PhysicSort (L\_ReturnFlag,L\_LineStatus)   
 PhysicVectorToRow (L\_ReturnFlag, L\_LineStatus)  
 PhysicVectorizedHashAgg (L\_ReturnFlag, L\_LineStatus)  
 PhysicRowToVector(L\_ReturnFlag, L\_LineStatus)  
 PhysicScanTable (LineItem + filter)

This means we must first convert rows into vectors, feed into hash aggregation, then convert them back to rows for sorting purpose. When the column store is provided, as it provides native vector format, we can directly scan it with the column store scan iterator:

PhysicSort (L\_ReturnFlag,L\_LineStatus)   
 PhysicVectorToRow (L\_ReturnFlag, L\_LineStatus)  
 PhysicVectorizedHashAgg (L\_ReturnFlag, L\_LineStatus)  
 PhysicScanCstore (LineItem + filter)

Here is a more complete example running against the columnar storage for TPCH Q3:

PhysicLimit (cost=0.07..0.08 rows= …)

-> PhysicGather (cost=0.00..0.00 rows= …)

-> PhysicLimit (cost=0.07..0.08 rows= …)

-> PhysicSort (cost=0.07..0.08 rows= …)

Sort Key: (sum((lineitem.l\_extendedprice \* …

-> PhysicVecToRow (cost=0.06..0.06 rows= …)

-> PhysicVectorizedAggregate (cost=0.06..0.06 rows= …)

-> Vector Hash Join (cost=0.03..0.05 rows= …)

Hash Cond: (orders.o\_orderkey = lineitem.l\_orderkey)

-> Vector Hash Join (cost=0.01..0.03 rows= …)

Hash Cond: (customer.c\_custkey = orders.o\_custkey)

-> PhysicScanCstore customer (cost=0.00..0.00 …)

Filter: (c\_mktsegment = 'BUILDING'::bpchar)

-> PhysicScanCstore orders (cost=0.00..0.00 …)

Filter: (o\_orderdate < '1995-03-15'::date)

-> PhysicScanCstore lineitem (cost=0.00..0.00 …)

Filter: (l\_shipdate > '1995-03-15'::date)

In the above plan, the tables are first joined with the vectorized hash join followed by the vectorized aggregation. After that, the vectors are converted to rows and further processed in the row engine with sorting and limit. Then the result is finally output.

**Ref**: OpenGauss vector engine implementation.

## Parallel Execution

There are two types of parallel query execution: inter-query parallelism and intra-query parallelism. Inter-query parallelism studies the independent streams of query execution and intra-query parallelism studies how to execute a single query in parallel. TPCC and TPCE benchmark are all about former: there are multiple connections to a database server and each connection issues a sequence of a SQL commands. The target is to maintain certain response time while maximize completed transactions in unit time. TPCH and TPCDS benchmarks are about both. They have a power run stage, which issues a single query each time and expect them finish in the shortest time. They also have a throughput run stage, which there are more than 10 streams with each stream issues a sequence of power queries. The target is to minimize the longest completion connection.

After a query is compiled into execution plan, the first thing query execution engine to do is to budget and assure resources, including available threads and memory. The major fact deciding how much resource to use is the DOP. First SQL Server try the desired DOP, calculate number of threads and memory amount for current query. When the calculation with simple checks is done, query entering the stage waiting for memory granted. If this waiting is timed out, the query just error out. User can restart the query when the system is not that busy. After the waiting is done, query engine will try to allocate resource for its execution. This can fail due to resource pressure as the system is dynamic running. Query engine will downgrade the DOP and put back this query to the stage of resource estimation. During above resource grant process, there are several policies can change the situation: user may set MAXDOP for the server or for this query, or user can create a resource government resource group to put more or less restrictions on the query resource quota. As we can see, the two most important resources for parallel execution are memory grant and thread management.

### Scale out

Executor scale out follows the classic remote exchange model and QPModel has a multi-threaded emulation of the implementation. To unbind executor and underlying storage engine, the optimizer shall be able to introduce a level of M:N redistribution. Scale out shall support two forms of communication schema. The push-based schema is good for small to medium cluster and the M/R based schema is good for massive scaled out cluster. The push-based schema has best performance but might suffer from intermediate communication errors on large cluster. QPModel c# emulates a push-based implementation.

**Geo-cluster**

Traditionally, many companies’ global businesses either only use a central database, or each region has its own database, most of the business occurs locally (95%), and then use middleware to handle transactions or queries across regions (5%). Take Amazon.com as an example. It has 13 global sites, and each site has its own independent user (users of different sites cannot log in to each other), product catalog, transportation system, and user reviews. This loosely coupled way matches the independent business model of each site (for example, station C can use WeChat to log in but station A does not, station C has Double 11 and station A is Black Friday, and the transportation system used by station C is completely different from station A. ) Provides independent performance and availability. Furthermore, if a globally consistent view is provided on these global sites, the headquarters can obtain real-time business data.

This shows that Geo-deplorability should be added as an expansion capability rather than a mandatory mechanism-because different regions have different services, mandatory is a burden instead. The basis of the existence of Geo-database is to give priority to local transactions. The bonus item is to provide a global unified view, eliminate transactions and query third-party middleware to make cross-region transactions and management easier. The benefits of this non-mandatory approach are:

* Unification: Geo-deployable database can provide a single global database view that satisfies database semantics.
* Compliance: Some regions require local data storage, and Geo-deployment needs to meet this.
* Performance and usability: Most visits are released locally, and local visits are faster than remote visits. The same is true for availability.
* Non-mandatory: The database of each region is independently operable, and most operations do not require global coordination. Therefore, on the one hand, it is necessary to support the same logical structure and physical heterogeneity of the tables (public table, globally visible) of different regions, and on the other hand, support the private table (private table, globally invisible) in the region. The former is used for sharing, the latter is used for exclusive. An initial global deployment may not even have public tables, and then gradually add private tables-this makes the database more acceptable.

We can punish the performance of GLOBAL transactions within a certain range in exchange for not being damaged for LOCAL transactions. This can be achieved by several possible decentralized transaction protocols: A candidate protocol is CLOCK-SI+2PC if current storage engine implements SI protocol.

## Other Topics

### Adaptive Query Execution

The optimizer essentially selects query plans based on inaccurate information, so fundamentally we must allow the optimizer's plan to be non-optimal, or even bad. This has led to the emergence of adaptive execution technology: (1) The executor needs to be able to adjust the execution plan and feed it back to the optimizer when it runs; (2) the optimizer needs to learn from the feedback. **Ref:** Spark.

### Query Execution with Resilience

One of main advantage of M/R method is that it is resilient to node error as it can restart with previously staged work without repeating it. This method however introduces non-trivial and blind overhead to normal query execution. Query optimizer understands the cost of each subtree in the plan the cost the of persists its result. Thus, it can generate a plan with reasonable overhead of persists selected subtree results while minimize repeated work during recovery. For example, for a subtree like this:

PhysicHashAgg cost 160  
 PhysicHashJoin cost 60（build）+ 80（probe）  
 PhysicScan(A) cost 70  
 PhysicScan(B) cost 200

In different stage of execution, say when build is half done, when build is done, when probe is half done, the optimizer can evaluate the cost of persists results vs. saved restart repeating cost and decide if persists current results is needed.

### Engineering Considerations

**Query Cancel Support:** There are multiple scenarios of query canceling, for example, transaction error, query time out, user-initiated query canceling.

**Stable performance**: Query stability and cliff problems will appear on the resource boundary (cache/memory/disk) and execution plan selection boundary, because the two sides of the boundary are different implementation and cost models, resulting in non-linear performance experience. At present, some specific problems (such as aggregation) have known methods to solve, but there is no general solution.

### Other Items

**Multi-modal Support**: Executor is high extensible with multiple purposed physic node or engine sit side by side, so multi-modal support is not a concern with executor level. **Ref:** OpenGauss.

**Modern Hardware**: Academic have many researches with OLAP will leverage bandwidth-oriented hardware and OLTP can further use latency-oriented hardware for acceleration.

The popularization of RDMA simplifies local clusters and single machines into one type of architecture. A common dispute is whether the cluster should be shared disk (SD) or shared nothing (SN)? Both SN and SD have multiple nodes. The essential difference is the ownership of the data: if a piece of data can only be accessed by one node (other nodes need to access it through the node agent), then this is the SN architecture, otherwise it is the SD architecture. There are many discussions about which one is better for SN and SD. Although SN is destined to be less efficient than SD in some cases (such as small query non-shard alignment), in essence, SN coordinates control rights, while SD coordinates data, and the amount of control rights is much smaller than the amount of data. So, we embrace the SN architecture.

# Storage Engine

## Design for OLTP

**Ref**: Molehill.

## Design for OLAP

**Ref**: OpenGauss.

# Appendix

## Spark DataSet API Example

Here is a simple DataSet application with Spark’s plan, where we shall observe how Spark supports it:

|  |
| --- |
| case class Dog(name: String, age: Integer) val ds = Seq(Dog("d1", 1), Dog("d2", 2)).toDS;  scala> ds.filter($"age">1).repartition($"age").sort($"name").rollup($"age").count().join(df, "age").cube("age").count().union(df.select($"age", $"name")).explain(); == Physical Plan == Union :- \*HashAggregate(keys=[age#369, spark\_grouping\_id#367], functions=[count(1)]) : +- Exchange hashpartitioning(age#369, spark\_grouping\_id#367, 200) : +- \*HashAggregate(keys=[age#369, spark\_grouping\_id#367], functions=[partial\_count(1)]) : +- \*Expand [List(age#368, 0), List(null, 1)], [age#369, spark\_grouping\_id#367] : +- \*Project [age#353 AS age#368] : +- \*BroadcastHashJoin [age#353], [age#11], Inner, BuildRight : :- \*HashAggregate(keys=[age#353, spark\_grouping\_id#351], functions=[]) : : +- Exchange hashpartitioning(age#353, spark\_grouping\_id#351, 200) : : +- \*HashAggregate(keys=[age#353, spark\_grouping\_id#351], functions=[]) : : +- \*Filter isnotnull(age#353) : : +- \*Expand [List(age#352, 0), List(null, 1)], [age#353, spark\_grouping\_id#351] : : +- \*Project [age#341 AS age#352] : : +- \*Sort [name#340 ASC NULLS FIRST], true, 0 : : +- Exchange rangepartitioning(name#340 ASC NULLS FIRST, 200) : : +- Exchange hashpartitioning(age#341, 200) : : +- \*Filter (isnotnull(age#341) && (age#341 > 1)) : : +- LocalTableScan [name#340, age#341] : +- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, true] as bigint))) : +- \*Project [age#11] : +- \*Filter isnotnull(age#11) : +- LocalTableScan [name#10, age#11] +- LocalTableScan [age#11, name#10]  scala> ds.filter($"age">1).map(x=>x.age+1).explain(); == Physical Plan == \*SerializeFromObject [input[0, int, true] AS value#536] **+- \*MapElements <function1>, obj#535: int**  +- \*DeserializeToObject newInstance(class $line40.$read$$iw$$iw$Dog), obj#534   +- \*Filter (isnotnull(age#341) && (age#341 > 1))  +- LocalTableScan [name#340, age#341] |

Here is a comparison between aggregation with cube:

tpch1g=# explain verbose select \*, sum(age) from dog group by cube(name, age) order by name;

QUERY PLAN  
--------------------------------------------------------------------------------  
 Sort (cost=168.92..171.92 rows=1202 width=96)  
 Output: name, age, (sum(age))  
 Sort Key: dog.name  
 -> GroupAggregate (cost=50.08..107.43 rows=1202 width=96)  
 Output: name, age, sum(age)  
 Group Key: dog.name, dog.age  
 Group Key: dog.name  
 Group Key: ()  
 Sort Key: dog.age  
 Group Key: dog.age  
 -> Sort (cost=50.08..51.83 rows=700 width=88)  
 Output: name, age  
 Sort Key: dog.name, dog.age  
 -> Seq Scan on public.dog (cost=0.00..17.00 rows=700 width=88)  
 Output: name, age

And here is the Spark plan:

scala> ds.cube("name", "age").sum("age").orderBy("name").explain  
== Physical Plan ==  
\*Sort [name#181 ASC NULLS FIRST], true, 0  
+- Exchange rangepartitioning(name#181 ASC NULLS FIRST, 200)  
 +- \*HashAggregate(keys=[name#181, age#182, spark\_grouping\_id#178],   
unctions=[sum(cast(age#3 as bigint))])  
 +- Exchange hashpartitioning(name#181, age#182, spark\_grouping\_id#178, 200)  
 +- \*HashAggregate(keys=[name#181, age#182, spark\_grouping\_id#178], functions=[partial\_sum(cast(age#3 as bigint))])  
 **+- \*Expand [List(age#3, name#179, age#180, 0), List(age#3, name#179, null, 1), List(age#3, null, age#180, 2), List(age#3, null, null, 3)], [age#3, name#181, age#182, spark\_grouping\_id#178]**  
 +- LocalTableScan [age#3, name#179, age#180]

1. A common controversy is whether the second level is good enough? Considering the end-to-end system recovery, including fault detection, reconnection attempts, etc., we believe that the second-level fault recovery is acceptable. [↑](#footnote-ref-1)