Compiler-based execution of SQL

Table of Contents

This is a short survey of systems which have been developed for compiler-based execution of SQL queries.

Overview

In database systems, two approaches have been developed for transforming declarative SQL queries to imperative execution.

Template-based :

Almost all commercial DBMSes implement a Volcano Iterator model for query execution. In the early versions of this model, one tuple at a time was passed through an entire tree of iterators.   This model was seen to be inefficient. Later versions of this model use a "block"-based approach and process a set of tuples at a time.

Iterator models can be pull (synchronous) or push (synchronous).

Other variants have included MonetDB and VectorWise which do vector operations that fit into the CPU cache.

Compiler-based model:

With the coming of large memory systems and in-memory databases, the inefficiencies of the previous model have become more exposed.   The iterator model prevents tight loop executions, and is unable to exploit the data cache or reuse CPU registers.

This spurs the move to compiler-based systems which can enable data-centric execution.

Just as in programming languages, a compiler does the following transformations *high level Language -> AST -> Intermediate language -> Machine code*

Similarly, an SQL query can be transformed as:

SQL -> AST -> Intermediate language -> Final code -> Compile/Load and run

Query compilation systems

A compiler-based system performs the following transformations

SQL -> AST -> Intermediate language -> Java/C/C++/LLVM assembly which is then loaded and executed.

Steno

(microsoft research, univ of cambridge)

LINQ (Language integrated query) is a declarative extension to C#. The Steno system takes a declarative query and generates a C# class for it which is loaded and executed by the system.

Steno does two major optimizations:

1. **Iterator fusion** : For example, it can combine a SELECT and WHERE operator into a "for loop".
2. **Transform nested queries into nested loops**

Transformation steps taken:

1. Given a chain of operators, it emits symbols of QUIL (query intermediate language).
2. From the sequence of symbols, it uses a FSM to generate a C# class using the CodeDOM library.
3. Compile and load this query for execution.
4. Before execution, it resolves any object references in the query by using the reflection API in C#.

QUIL
The QUIL language has six symbols (SRC, TRANS, PRED, SINK, RET, AGGR).
The sequence of symbols starts from SRC and ends in RET.
SRC signifies the start of execution and represents a table or index scan which creates a list for processing.
TRANS, PRED and SINK transform one list to another.
RET returns the final list.

How is code generated ?

On a SRC symbol, it generates a "for loop" based on the symbol arguments. This 'for loop' is represented as a linked list of 3 nodes (precursor, loop body, and post-operation).
On a TRANS symbol, it generates element-wise operations inside the loop body
On a AGGR symbol, it reduces the list to a scalar value
On a SINK symbol (i.e. a "group by" or "order by" clause). For a "group by", it creates a new list

from the old using an intermediate hash table or tree. For "order by", it invokes a C# sorting function
On a PRED symbol, it generates an "if clause".

Example:
If the query was "*SELECT max(a) from <table1>*"

then the sequence of symbols will be "*{SRC <table1>, AGGR <a>, SINK }*"

and the code generated will be :

```
foreach (elem in table1)
{
    max_a = max(elem, max_a)
}
```

Distributed query

To generate a distributed query,

given a sequence of QUIL symbols { SRC, TRANS, AGGR, RET}

Steno/DryadLINQ inserts intermediate symbols which expand the query to all nodes and aggregate the results (kind of like a map-reduce)

```
SRC -> SRC (node1) -> TRANS(node1) -> AGGR(node1) -> AGGR -> RET
    -> SRC (node2) -> TRANS(node2) -> AGGR(node2) ->
```

Krikellas – holistic query evaluation

(univ of edinburgh)

It takes the output of the query optimizer, which is a topologically sorted list of operations and applies them over pre-defined code templates, parametrized by machine architecture, and uses it to generate code.

The time for generating the code was found to be about 25 ms. The time for compiling optimized code was about 400-600 ms.

Challenges they identified:

1. identify common code templates
2. how to interconnect different operators now that they have no common API
3. how to verify correctness of the generated code.

HyPer system – Neumann

(Tum.de)

This is positioned as a hybrid OLTP-OLAP database (academic research)

First they tried to generate C++ code from an SQL query. Later they switched to generating LLVM assembly code, which they found was equally manageable and efficient. Compiling the LLVM assembly to the final machine code is faster than the same for C++.

They use LLVM code only to replace the tight inner loops which occur during query execution.

Hekaton (SQL server in memory db)

They realized that the type systems and expression semantics of T-SQL and C are very different.

1. T-SQL includes many data types such as date/time types and fixed precision numeric types that have no corresponding C data types.
2. In addition, T-SQL supports NULLs while C does not.
3. Finally, T-SQL raises errors for arithmetic expression evaluation failures such as overflow and division by zero while C either silently returns a wrong result or throws an OS exception that must be translated into an appropriate T-SQL error.

That is why they introduced an additional step in code generation which generates a “Pure Imperative Tree” from the “Mixed Abstract tree”.

DBToaster

Steps

1. Parser
2. Algebraic compiler : They keep a map algebra of about 70 simplification rules which get applied to a query.
3. Code generator : Generate C++ code.

Apache Spark (optimizer is called Catalyst)

Other query optimizers employ a domain-specific language to define rules and write a custom compiler to generate code. By contrast, Catalyst exploits Scala language features such as pattern-matching and quasiquotes for the same purposes. It uses the Java compiler to gnerate bytecode which will execute on Spark nodes.

It does both cost-based and rule-based optimization. (Cost-based means it chooses the plan with the least cost; rule-based means it applies pattern matching rules to transform the query execution tree).

Steps

1. Convert AST to logical plan
2. Analyzing a logical plan to resolve variable references (i.e. lookup the schema)
3. Logical plan optimization,
4. Physical query execution plan

5. Generate Java bytecode using Scala **quasiquotes** feature

In Scala, when you prefix a string with a "q" (i.e. val mytree = q"this is a tree"), the Scala compiler internally stores the variable "mytree" as a Syntax Tree rather than a string.

http://docs.scala-lang.org/overviews/quasiquotes/intro.html

Now you can write Scala code to compare trees or transform them. Catalyst uses this feature to store query execution trees and generate Java bytecode from them.

https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html

Apache Tajo

Tables are stored in columnar format and bytecode for vectorized primitives is generated at runtime.

They use the unsafe Java library (sun.misc.unsafe) to do direct in-memory optimizations on vectors. Memory allocated via "unsafe" is not under GC control and not constrained by JVM heap size.

http://www.slideshare.net/Hadoop_Summit/t-435p210-achoiv2

MemSQL

Produces C++ code which is bundled into a shared library.

VitesseDB (Postgres enhancement)

They run the Vitesse JIT engine after a query plan is created.

They got 2X performance by applying LLVM to compile the expressions found within the query, while the Plan tree was still interpreted using the iterator model.

They got 8X performance by compiling entire query into one JIT procedure. The query nodes got inlined, CPU registers were reused and code had tight inner loops.

They obtained 108X performance by running on multi-core and letting each core work on a distinct data set.

They obtained 180X performance by using a columnar data store with LZ4 and Delta compression.

http://vitessedata.com/

Slides here: http://goo.gl/Mtg2W6

Facebook Presto

It dynamically compiles certain portions of the query plan to Java byte code.

See the **ExpressionCompiler** class which in turn uses **PageProcessorCompiler** and **CursorProcessorCompiler**

https://github.com/facebook/presto/blob/c0e32e0e9a787708250f3c47f8f11567cbe679ae/presto-main/src/main/java/com/facebook/presto/sql/gen/ExpressionCompiler.java

This class is invoked from the LocalExecutionPlanner

Cloudera Impala

It uses LLVM to compile stored procedures into LLVM IR (intermediate representation) language.

http://llvm.org/devmtg/2013-11/slides/Wanderman-Milne-Cloudera.pdf

References

1. D. G. Murray, M. Isard, and Y. Yu. Steno: Automatic Optimization of Declarative Queries. In PLDI, 2011.
2. K. Krikellas, S. D. Viglas, and M. Cintra. Generating code for holistic query evaluation. In ICDE, 2010.
3. T. Neumann. Efficiently compiling efficient query plans for modern hardware. Proc. VLDB Endow., 4(9), 2011.
4. C. Diaconu, C. Freedman, E. Ismert et al., “Hekaton: Sql server’s memory-optimized OLTP engine,” in SIGMOD ’13, 2013.
5. Armbrust, et al . Spark SQL: Relational Data Processing in Spark. SIGMOD 2015 http://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf