Compiled Query Execution Engine using JVM

Jun Rao, Hamid Pirahesh, C. Mohan, Guy Lohman

IBM Almaden Research Center
{junrao,pirahesh,mohan,lohman}@almaden.ibm.com

Abstract

A conventional query execution engine in a database system essentially uses a SQL virtual machine (SVM) to interpret a dataflow tree in which each node is associated with a relational operator. During query evaluation, a single tuple at a time is processed and passed among the operators. Such a model is popular because of its efficiency for pipelined processing. However, since each operator is implemented statically, it has to be very generic in order to deal with all possible queries. Such generality tends to introduce significant runtime inefficiency, especially in the context of memory-resident systems, because the granularity of data processing (a tuple) is too small compared with the associated overhead. Another disadvantage in such an engine is that each operator code is compiled statically, so query-specific optimization cannot be applied.

To improve runtime efficiency, we propose a compiled execution engine, which, for a given query, generates new query-specific code on the fly, and then dynamically compiles and executes the code. The Java platform makes our approach particularly interesting for several reasons: (1) modern Java Virtual Machines (JVM) have Just-In-Time (JIT) compilers that optimize code at runtime based on the execution pattern, a key feature that SVMs lack; (2) because of Java's continued popularity, JVMs keep improving at a faster pace than SVMs, allowing us to exploit new advances in the Java runtime in the future; (3) Java is a dynamic language, which makes it convenient to load a piece of new code on the fly. In this paper, we develop both an interpreted and a compiled query execution engine in a relational, Java-based, in-memory database prototype, and perform an experimental study. Our experimental results on the TPC-H data set show that, despite both engines benefiting from JIT, the compiled engine runs on average about twice as fast as the interpreted one, and significantly faster than an in-memory commercial system, using SVM.

1 Introduction

Conventional database execution engines [10] evaluate a query by interpreting the query execution plan (QEP) generated by a query planner (or optimizer). A typical QEP is a tree structure in which each node corresponds to a relational operator such as scanning a table or an index, selection, projection, join, aggregation, etc. During query evaluation, each node consumes data produced by its child/children, one tuple at a time, and generates output of its own, also a single tuple at a time. Such an interpreted execution engine makes pipelined operations very efficient, but pays an overhead at runtime. The main deficiency comes from the fact that the implementation of each operator is static, and thus has to be written in a very generic way to accommodate queries of any kind. For example, no assumptions can be made on such things as the number of columns being accessed and the type of each of those columns.

To understand the problem better, let's consider a concrete example. Suppose that the SQL of a query Q1 is given by "select X from T where Y > 10", where Y is an integer column. Also suppose there is no index on column Y, so that the predicate has to be evaluated by scanning every tuple in table T. Let's consider how a predicate evaluator is built in an interpreted engine. There are many types in SQL on which the ">" predicate could be applied, such as short integer, integer, long integer, float, double, character, string, etc, each of which requires a different implementation of ">" (some systems have separate versions for each nullable type). A generic predicate evaluator, not knowing beforehand which version of ">" should be applied, normally has to build a function array storing the implementation of all possible versions, and invokes the predicate evaluation through a function pointer. Typically, a generic void* is used for input arguments to this function and data is cast to the right type inside each function. The function pointer is only initialized at runtime when a QEP is given, which makes it impossible to inline the actual ">" function used. Thus, for a simple comparison of two integers, typically done in one cycle on modern processors, an overhead of making a function call has to be paid. A typical function call involves pushing inputs (which potentially requires memory accesses) onto the stack, jumping to the function, and then popping up the return values from the stack. This takes about a few tens of cycles itself, and may slow other operations down because of the extra memory accesses.

Such an overhead has been more or less ignored in conventional disk-based systems because I/O costs and the cost of navigating through the disk-oriented data representation often dominate CPU costs. However, it could be significant in a memory-resident database system in which there is no I/O cost for reads and the data layout is optimized for main memory not disk. Main memory is getting cheaper and larger. Currently, 1GB of RAM costs less than $300. An IBM Power5 570 server can support up to half a terabyte of RAM on a single machine (although right now a high premium has to paid for the maximal memory capacity), enough to hold all data and index structures in memory for a large number of applications. Bigger databases can be handled by a cluster of machines configured with tens of terabytes of main memory. Research in main memory database systems has made in-memory data processing more and more efficient. For example, by exploiting pointers to store join relationships, join processing can be as simple as following those pointers. As another example, cache-conscious indexing structures make index traversal much faster in memory.

We argue that a conventional interpreted query execution engine is severely inefficient in the context of a main-memory database system, and propose a new "compiled" execution engine to address this problem. In contrast with an interpreted engine, a compiled engine generates new operator code for a given QEP dynamically at runtime. Since a QEP provides all the information about a query, a specialized piece of code can be generated, avoiding the overhead of a generic implementation. In the previous example, the comparison instruction can be inlined into the scan operator. The new piece of code then gets compiled and loaded for query evaluation. The benefit from specialization should offset the extra compilation and loading overhead, especially for queries statically compiled once and executed many times (possibly with different bindings).

This paper evaluates the benefits of a compiled engine in the context of Java, which has gained significant momentum in the application development domain since it was invented by Sun Microsystems in 1995. There are a number of reasons why Java is very appealing. First of all, Java compilers are dynamic. A Java Just-in-time (JIT) compiler compiles Java bytecode into machine code on the fly based on the information from a pilot run, and thus is capable of producing more efficient code than a static compiler such as C. There is an interesting analogy here. A Java program used to be interpreted by a JVM directly (like the interpreted SQL engine). At present, the preferred way of evaluating a Java program is to enable JIT to generate specialized code on the fly. We believe that such an approach should be taken by an SQL engine, as well. Second, by generating specialized Java code (or bytecode) on the fly, we rely on Java to provide low-level code optimization, which keeps improving due to the heavy investment by the large Java community. Finally, Java is dynamic. Java reflection provides a mechanism of loading new classes (through Class.forName()) and methods by name. In contrast, in C/C++, one needs to manipulate dynamically-linked libraries in order to execute a newly generated piece of code.

The rest of this paper is organized as follows. We provide some background information and summarize related work in Section 2. In Section 3, we revisit the conventional database execution engine and previous attempts at making it more efficient. In Section 4, we describe JAMDB, a relational, in-memory, Java-based database prototype, which we use to quantitatively evaluate an interpreted engine and a compiled one. We present experimental results in Section 5, where we show a compiled Java-based engine runs significantly faster than interpreted engines in both Java and C. We discuss future work in Section 6 and conclude in Section 7.

2 Background

In this section, we discuss the background related to this paper: Java technology and main memory databases. We also summarize the related work in these two topics.

2.1 Java Technology

Java is a dynamic, type-safe, object-oriented programming language. A Java program is compiled to platform-independent bytecode, which is then executed by a Java Virtual Machine (JVM). A JVM provides many built-in OS-style capabilities such as threading, synchronization, and memory management. Historically, Java has been considered slow because the bytecode was originally interpreted by the JVM. Also, since Java is a "safe" language, it introduces runtime overhead such as array boundary checking and type conversion validations. However, Java performance has significantly improved in the last few years. Since the introduction of Just-In-Time (JIT) compiler technology, a JVM can compile a piece of bytecode into machine code the first time it is executed. Subsequent executions of the same bytecode use the compiled version to run much faster. When generating the machine code, a JIT compiler can perform dynamic optimization based on the execution pattern of the code. Since JDK 1.4, Sun has added the HotSpot capability, which improves JIT by focusing compilation optimization on those segments of the bytecode taking most of the time. As a result, HotSpot is able to apply more optimization at runtime. Because of JIT compilation, Java has become very competitive with C/C++. For instance, [13] reports that for certain scientific benchmarks, Java is comparable to C in performance. Similar promising performance results [25] have also been reported on the .NET platform, which is very similar to a JVM, but limited to Windows. In Figure 1, we compare the performance of adding ten million integers together in Java and C (all compilers are IBM's) on an AIX 5.2 Power 4 machine. As we can see, Java performance keeps improving relative to C, and with the latest version, is within a factor of two of the C program (in fact, we had to turn on the highest optimization level of the C compiler in order to beat Java). In theory, a Java program can run faster than C/C++, because JIT compilation happens dynamically and can take advantage of the actual code path taken. A C/C++ compiler is static and does not take into account any runtime information.

	Java 1.3	Java 1.4	C
Time (s)	0.12	0.03	0.016

Figure 1 Time to add 10 million integers

Because of historical performance concerns, the database community in general has not adopted Java as a serious platform for building data-intensive servers, with a few exceptions. The Derby open source DBMS [14] and Cloudscape [31] are pure Java relational database management systems. A Java-based database server for streaming applications is described in [26], and its pros and cons of using Java are shared. However, both systems are disk-based, and, as far as we know, use an interpreted query execution engine. Both Derby and Cloudscape generate Java bytecode for query execution. However, the generated bytecode is not specialized for each query.

Because of Java's dynamic nature, the idea of generating specialized Java code on the fly has been explored in other domains. For example, Apache Xalan XSLTC [38] compiles an XSL stylesheet into a translet, which is a set of Java classes, and then uses a runtime processor to apply the translet to an XML document to perform a transformation. Jython [33] is an implementation of the high-level, dynamic, object-oriented language Python, seamlessly integrated with the Java platform. It provides high performance by dynamically compiling Python code into Java bytecodes.

2.2 Main Memory Database Systems

Research in memory-resident database systems started as early as the mid-eighties, and has becomes a hot topic again in the past few years because large amounts of main-memory are increasingly available and affordable. Main-memory database systems are available both as research prototypes and commercial products, such as Dali [15] by AT&T, MonetDB [20] by CWI, and TimesTen [35], recently acquired by Oracle. A lot of previous research recognized the fact that memory accesses and CPU cost dominate query processing, in a memory-resident system, and developed new techniques to improve performance in those areas. MonetDB [5] and [2] exploit a column-wise storage layout (either for the whole table or for a data page) to reduce the amount of memory accessed by queries requesting only a small number of columns. [22,23] describe new index structures that reduce the number of hardware cache misses for index traversal. A new radix-based join method is proposed in [18] to further reduce the CPU cost of in-memory joins. [11,38] consider scheduling the relational operators across or within queries differently to minimize instruction cache misses. All these efforts make the overhead inherited in interpreted engines more and more significant, relative to "real" work. As we will see later, a compiled query execution engine both improves memory access pattern and reduces CPU overhead, and is generally applicable to any type of SQL queries.

3 Query Execution Engine Revisited

Most relational database engines use interpreted query execution engines that follow the dataflow model described in Volcano [10]. Each relational operator implements an open()-next()-close() interface, where open() initializes the internal data structures, next() computes and returns the next result tuple, and close() does the cleanup. In such a model, only a single tuple is passed among different operators. This introduces a lot of overhead in processing simple arithmetic and Boolean expressions, which are common in database applications.

Such overhead in an interpreted engine has been observed by earlier work [7,21], which tries to address the problem by increasing the granularity of work during pipelining. MonetDB/X100 [7] proposes a hyper-pipelining model that passes multiple tuples at a time among operators. The runtime engine is extended with vectorized operation primitives that accept an array of values as inputs. Although function calls are still needed, they are now much fewer and their overhead is amortized. However, such an approach typically introduces another problem for pipelining. An efficient implementation (used in most commercial systems) of the dataflow model associates each accessed column with a buffer holding exactly a single value, and shares buffers among pipelined operators, reducing the overhead of data movement. For example, consider a pipelined left-deep plan. The buffers for each column referenced (call them B) associated with the outermost operator (assuming a scan) are shared by all subsequent operators. Each time the scan operator fills in B with a new tuple T, the values in B are reused for each join result (both intermediate and final) T generates, without extra copying. In contrast, it is difficult to achieve that efficiency with hyper-pipelining. Every time T gets multiplied (because of joins), hyper-pipelining needs to duplicate values in the vectorized buffer. Although pointer copying (instead of value copying) is possible, for smaller data types, the overhead is almost the same. Such additional overhead with hyper-pipelining may well offset the benefit of amortizing function calls. It is for precisely this reason that [21] limits the vectorized processing to only blocking operators such as sort. Nevertheless, significant improvement is reported in [21] on certain

TPC-H queries, when vectorized processing is applicable. It's possible to exploit vectored processing to further enhance a compiled execution engine.

Most commercial database systems are implemented in C/C+. The drawback of the engine being statically compiled has also been observed. Some commercial database systems rely on feedback-based post-link optimization tools to address this issue. Feedback Directed Program Restructuring (FDPR) [30], developed by IBM, is such an example. FDPR takes the binary executable of a system (typically generated by a C/C++ compiler) and runs it through a training workload. Based on the feedback collected, FDRP is able to identify pieces of code that are critical. It then post-optimizes the binary through code repositioning and reordering. The AIX versions of IBM DB2 are post-optimized by FDPR and typically obtain an improvement between 10% and 15%. While FDPR is very good at optimizing code common to a workload (such as error handling), it is hard to do the specialized optimization for individual queries.

The idea of generating code on the fly for query evaluation has been explored before. System R [4] had a code generator that transformed a QEP into specialized assembly code on the fly. Several issues arose from generating assembly directly, however. The code generator was relatively hard to maintain and to debug. Also, the code generator had to implement low-level optimizations typically done by a compiler of a high-level language, such as loop unfolding and common subexpression elimination. More recently, Daytona [9], a data management system developed and used by AT&T, translates its high-level query language CymbalTM (including SQL as a subset) completely into C and then compiles that C into object code. Typically, the C code is generated beforehand for a given workload and then deployed to the actual system. Daytona avoids some of the issues in System R because it generates code in a high level language. However, since C code is compiled statically, dynamic optimizations performed by JIT compilers are not applicable. Neither System R nor Daytona provided a quantitative assessment of the benefit of a compiled execution engine. Finally, we note that generating code for better Boolean condition evaluation has been considered in a main memory database in [24] and the compiler community has explored code specialization in program compilation [8].

4 A Java-based Main Memory Database

In order to perform a quantitative analysis of the interpreted and the compiled engines, we implemented both engines in JAMDB, a Java-based main-memory database prototype, whose overall goal is to investigate exploiting modern JVMs as data service providers. In this paper, we focus on the querying aspect, since JAMDB at present is primarily a read-only system. In Section 6, we discuss possibilities of making such a system persistent.

Currently, JAMDB can process single-block SPJ SQL queries with aggregations. Nevertheless, it supports such complexity in SQL as arbitrary arithmetic and Boolean expressions, grouping expressions and null semantics. The overall architecture of JAMDB is depicted in Figure 2. At its core is an in-memory data store where data are represented as Java objects directly. A query first gets parsed to an internal query representation, which is then optimized by the query planner. We developed a simplified query planner that generates a left-deep QEP. It can be sent either directly to an interpreted engine for evaluation or to a dynamic code generator to produce a new specialized Java class that can be executed by the JVM. In the next three subsections, we describe data store, query planner, and runtime engines, respectively.

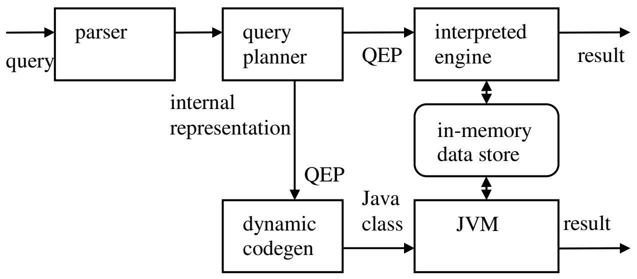


Figure 2 JAMDB Architecture

4.1 Data Storage

To fully exploit JVM, we store data as native Java objects and rely on Java to do the physical layout and memory management (Java garbage collector has improved significantly with the generational version, in which different stages are used to manage objects of various life expectancy). Storing data as Java objects reduces the overhead of conversion between the data storage and the execution engine, and is especially attractive when data resides in memory most of the time. Previous work [1,5] has argued for column-wise storage, since queries often ask for a small subset of the columns in a table. We choose to store data row-wise because column-wise storage is mostly beneficial for full table scans, and we expect to rely on index-based processing most of the time. The data store is depicted in more detail in Figure 3.

We store each tuple as a Java object. Given a table schema, two new Java classes are dynamically created, a tuple record class (R) and a table Class (T). Each class member in R is mapped from a column. Currently, four data types---int, double, date, and string---are supported. Both int and double are mapped to the corresponding Java primitive type. Date is stored as a 4-byte integer, where year, month, and day use 2, 1, and 1 byte, respectively. We follow the approach used in [18] by storing all unique string values in a column in a separate domain, and keeping only the integer identifier of the string value in R itself. Such an approach saves space and makes string equality comparisons cheaper. A hash table is built on the domain for fast string lookups (to obtain the identifier). A

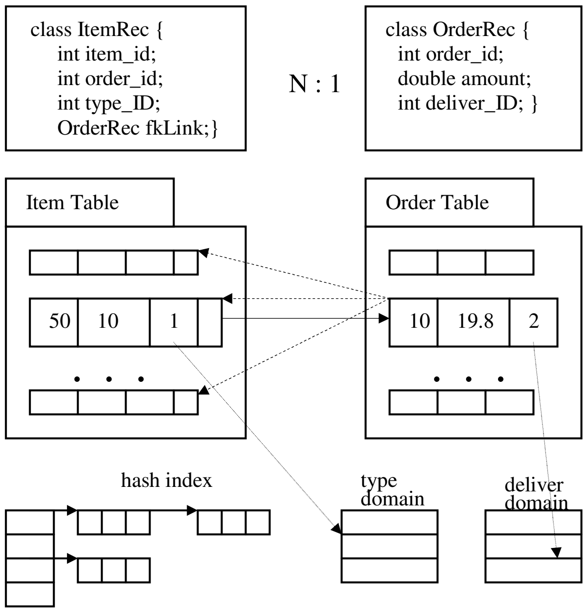


Figure 3 Data Storage Layout

class T is then created that stores tuple objects of type R in an extensible array (implemented as a two-dimensional array). Each class R also has a TID field, representing the internal tuple identifier (not shown in Figure 3).

We support both local indexes and join indexes. Local indexes are limited to a single column at this moment. Join indexes are maintained based on primary-key and foreign-key constraints. We support shared dimensions and allow a table to be referenced by multiple tables. For example, in Figure 3, there is an N-to-1 relationship between the Item table and the Order table. We expand the definition of the ItemRec class by adding a field (fkLink) to store the matching tuple in the Order table. A separate index on the TID field of the Order table is used to store the matching Item tuples (shown as the dashed lines in Figure 3). We implement chained-bucket hash tables as indexes because of their simplicity. To support range predicates for data types such as date, we choose order-preserving hash functions for local indexes defined on them. Each index entry stores a <key, tuple list> pair, where the tuple list contains references to tuple objects. We also keep in the index structure some basic statistics for query planning, such as the number of key values and the minimal and the maximal key values. In this paper, we do not deal with the index selection problem, but simply assume that any column having a predicate on it will have an index defined on it (can also use an index advisor).

Data are loaded in memory by calling the tuple insertion interface supported by each table class. Both local and join indexes are maintained during insertion.

4.2 Query Planning

We focus on simplicity when building the query planner. Because there is less penalty for random access in a main-memory database, index-based query processing becomes more competitive. In JAMDB, we limit the plan search space to left-deep trees, and all joins are evaluated by following the precomputed join indexes (join predicates other than PK-FK are treated as residuals).

Now we describe how to choose the join order. For a given join query, we define an anchor table X as the one from which all other tables can be reached by following one or more fkLink pointers from X, i.e., by following only the N:1 relationships. Since following references is cheap, we view all predicates as local to X. Join order selection then reduces to predicates ordering (ignoring the implied PK-FK predicates). Choosing the evaluation order of K local predicates has been studied before in the context of expensive predicates [12]. The optimal ordering is given by sorting all predicates by their ranks, where rank=(selectivity-1)/cost. To use this approach, we first estimate the selectivity of each predicate as relative to the anchor table X, by factoring in the domain size difference between the primary keys and the foreign keys. We then make a simplification by treating the cost of each predicate to be the same, except that when two predicates have the same selectivity, the predicate accessing the table with more PK-FK links to the anchor table is considered more expensive. The planner then makes a choice between two strategies. If the smallest selectivity exceeds a preset threshold, it picks a plan that starts with a table scan of the anchor table and evaluates the predicates in rank order by following fLinks. We refer to this type of plans as "scan plans". Otherwise, the planner picks the table with the most selective predicate as the outer-most table in the left-deep tree and uses a local index scan to retire that predicate. The rest of the predicates are applied as residuals in rank order by following the join indexes. This type of plans is referred to as "index plans".

4.3 Interpreted and Compiled Execution Engine

Interpreted Engine (IE) We implemented an interpreted query execution engine by following the tuple-at-a-time dataflow model. To be fair, we optimized IE performance as much as we could. First, we specialized its implementation whenever possible. For example, an implementation with null semantics is always separated from the one without. Second, we observed that Java reflection is expensive. In Table 1, we compare the cost of adding 10 million numbers together through inlining, 10 million interface (virtual function) calls, and 10 million reflection calls (Method.invoke()). Reflection is 20 times more expensive than interface. Reflection calls are very generic (arguments passed in through an Object array), and have the overhead of type casting (which is more than C) and indirect accesses to each argument. In our implementation, reflection is only used for class finding and all function calls are through interfaces.

	Inlined	Interface	reflection
Time(s)	0.038	0.112	2.63

Table 1 Time to add 10 million integers together

IE consists of three main types of statically implemented runtime operators: iterators, expression evaluators, and groupbys. All iterators support the standard open()-next()-close() interface. There are three types of iterators: scan, index, and join. A scan iterator sequentially scans through records in a table and returns them one at a time. An index iterator returns matching records for a predicate using a local index defined on a table. A join iterator handles a 1:N join (N:1 joins are handled by column fetching discussed in the next paragraph). It has a join index stacked on top of an input iterator, which is either an index or another join iterator. Each object returned by the input iterator is used for lookups in the join index, and the matching records are returned one at a time.

All expressions support an "evaluate" interface. There are many different classes of expressions. A column expression is responsible for fetching a column value given a record object from the iterator. Since IE does not know which column to fetch beforehand, fetching has to be virtualized. Each table has, for each of its data members D, a getD inner class supporting a "fetch" interface, which is implemented by casting the input record object to the right type and then copying its D value to an input expression. Each column expression identifies the right getD class through reflection at planning time and uses its fetch interface to obtain the data value at runtime (note there is no reflection cost for each fetch). We further implement a getX_D inner class for each table class, reachable from another table class X by following any number of fkLinks. This favors IE because, from a record in an anchor table, it can get a column value in another table in a single function call. If a table is (indirectly) referenced by another table multiple times through different paths (e.g., in TPC-H, lineitem table indirectly references nation table through both supplier and customer), we include all table names on the reference path in the inner class name.

We implement a number of arithmetic expressions to support binary arithmetic operations such as +, - and *, and a set of Boolean expressions for simple comparisons such as =, != and <, as well as more complex ones using conjuncts and disjuncts. A family of aggregate expressions are defined to support different aggregate functions on different types. There are some other miscellaneous expressions for constructs such as "case" and built-in functions (e.g., year()) on the date type.

Grouping is computed by a groupby operator using hashing. It takes as input an array of expressions as the grouping set. For each input tuple, it finds the right set of aggregate expressions to evaluate by searching through the hash buckets. Interfaces on the grouping expressions are used to compute the hash code and to compare two grouping values. A QEP for IE is essentially a tree of iterators, expressions and a groupby operator, each created properly based on the query and data types, and all linked together according to the data flow.

Compiled Engine (CE) Writing a dynamic code generator can be tricky. We want to balance between the efficiency of the generated code and the complexity of the generator. In JAMDB, we wrote the generator manually. We discuss in Section 6 the automation of this process.

The code generator in CE traverses the QEP (same in IE) bottom up. For each operator encountered, it generates new code (in general specialized) based on the operator type. The only specialization we do for iterators is on a scan. The scan operator already has information about which table is being scanned. Therefore, new code is generated such that all records in the table are converted to the right type once (using array casting). This way, during iteration, individual records no longer need to be cast and can be used directly. As we will see in Section 5, such an optimization can reduce runtime overhead significantly. We choose not to specialize the code of the index and join iterators because of code complexity. The generator simply reuses the same code that is in IE.

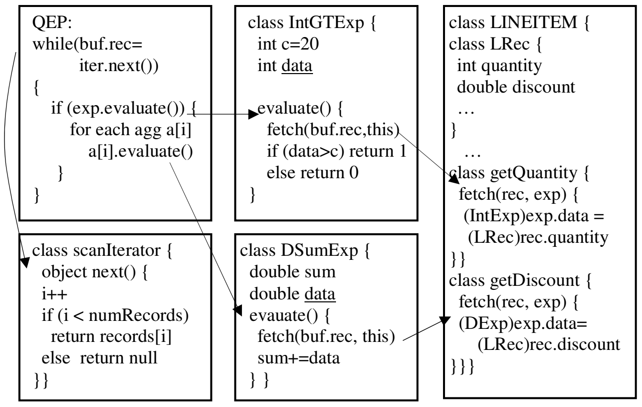
We specialize all expression evaluators. For a column expression, a direct reference to the corresponding data member of a record is used. This avoids the overhead of the fetch interface. For any other types of expressions, an inlined representation for the whole expression is generated to avoid the overhead of all virtual function calls. The same inlining is done for the groupby operator when comparing grouping values in the hash chain. We further specialize the code by enumerating the grouping and the aggregate expressions explicitly, instead of through a generic array structure. A new structure is also defined for storing aggregate results specific to the query.

Once all the operators are traversed, the code generator merges all pieces of generated code into a single block and produces a new query class implementing an "execute" interface. CE then makes an external call to the Java compiler to compile the new code into bytecode and loads the class into the JVM through reflection. Finally, the query is executed using the execute interface. Although not pursued in JAMDB, it's possible for a CE to generate Java bytecode (instead of Java source code) directly using libraries such as [16]. This way, a Java compiler does not need to be embedded in JAMDB.

An Example: We now illustrate the difference between IE and CE using the following SQL query.

```
Q1.  select sum(l_discount) from lineitem
     where l_quantity > 20
```

The pseudo code for the QEP for Q1 is shown in the left top box in Figure 4(a). The arrows illustrate the function calling sequence. The QEP first makes a call to the scan iterator to obtain the next record (for simple presentation, we assume all records are stored in a single array). The record is then put into a shared buffer. QEP then evaluates the Boolean expression corresponding to the selection predicate in Q1. That call is routed to a class (IntGTExp) implementing ">" on integer types. The Boolean evaluator first calls the proper fetch interface to



```
QEP:
while(buf.rec=
        iter.next())
{
   if (exp.evaluate()) {
      for each agg a[i]
          a[i].evaluate()
   }
}
```

```
class IntGTExp {
  int c=20
  int data

  evaluate() {
    fetch(buf.rec,this)
    if (data>c) return 1
    else return 0
  }
```

```
class LINEITEM {
class LRec {
  int quantity
  double discount
  …
}
  …
class getQuantity {
  fetch(rec, exp) {
    (IntExp)exp.data =
     (LRec)rec.quantity
}}
class getDiscount {
  fetch(rec, exp) {
  (DExp)exp.data=
       (LRec)rec.discount
}}}
```

```
class scanIterator {
  object next() {
  i++
  if (i < numRecords)
    return records[i]
  else  return null
}}
```

```
class DSumExp {
  double sum
  double data
  evauate() {
    fetch(buf.rec, this)
    sum+=data
  } }
```

(a) Interpreted Plan

```
class Q1 {  class Q1Agg { double sum1; }
              execute(){
                 Lrecords = (LRecord[])records;
                 while (i < numRecords) {
                    rec = Lrecords[i];
                    if (rec.quantity > 20) q1agg.sum1+=rec.discount;
                    i++ ;
            }}}
```

(b) Compiled Plan

Figure 4 An Interpreted and a Compiled Plan for Q1

obtain the data value from the record in the buffer (shared with the one in QEP). Explicit type conversion is required at fetch because the input type is generic (note that IntGTExp is a subclass of IntExp, from which the underlined "data" is inherited). Once the computed Boolean result is returned to QEP, it iterates through each aggregate expression (although there is in fact only one) and invokes its evaluation method. The call is passed to a class (DSumExp) that knows how to handle sum on double types. The evaluate function again fetches the data value first and then does an addition.

The functionally equivalent code generated for the compiled plan is shown in Figure 4(b). The conversion of the record type is done once for the whole table at the beginning. Both the Boolean expression and the aggregate expression are inlined. A special aggregate structure is created to store the single aggregate result. Compared with the interpreted plan, the compiled one is more efficient for three reasons. First, because of inlining, the overhead of casting and method invocation is mostly avoided. Second, a lot of the data passing can be done using registers, which reduces the number of potential memory accesses. Last but not least, the specialized code produces a much larger block of code for the JIT optimizer to work on. This creates a situation that significantly increases the effectiveness of compiler optimizations because more optimization techniques can be applied. It's hard for the JIT optimizer to perform the same optimization for the interpreted plan, in which code is separated in difference places.

5 Experimental Results

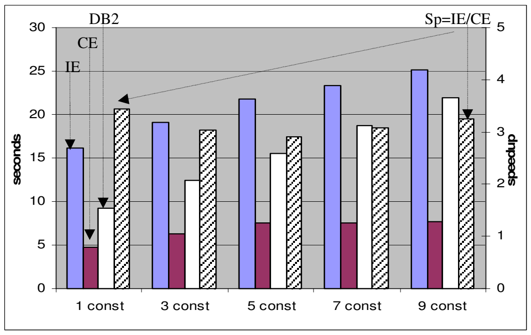
We developed JAMDB using Eclipse [29], an open source integrated development environment for Java. We tested JAMDB on two platforms. One platform (called AIX) runs IBM AIX 5.2 on an IBM P650 machine, which has eight 1.45GHz Power4 CPUs and 16GB of main memory. The JVM on AIX is 64-bit IBM J2SE 1.4.2 with JIT, which allows Java heap size much larger than 2GB. The second platform (called Windows) runs Windows 2000 on a PC machine, which has one 2.2 GHz Pentium IV CPU and 1.5GB of main memory. We chose two Java compilers on Windows, Sun Java J2SE 1.4.2 with HotSpot and IBM J2SE 1.4.2 with JIT, both of which are 32-bit versions. Because the results on the two JVMs are similar, we primarily present the results using the IBM JVM. We used data from the TPC-H benchmark [36] as the testing data set. The scale factor was selected to be 5 on AIX and 1 on Windows. For comparison, we also tested the performance of all queries on a commercial DBMS, IBM DB2 UDB V8.2 (other studies have shown that Java-based interpreted engines like Cloudscape currently do not perform better than DB2) on both platforms (the AIX version of DB2 has 64-bit support). DB2 uses a conventional interpreted runtime engine, and is written in C/C++. We configured DB2 with a large bufferpool to make all data and indexes memory-resident. To approximate the absence of locking in JAMDB, we used the lowest isolation level (UR) when running each query. We let DB2's optimizer choose the best execution plan, but set configuration parameters used to estimate disk seek overhead and transfer rate to very low numbers (the plans obtained this way are often better). In both JAMDB and DB2, the same set of local indexes is created. Since DB2 currently does not have join indexes, it uses B+-tree indexes defined on primary-keys for nested-loop joins. We ran each query three times and report the minimal elapsed time. The time for the interpreted plan, the compiled plan in JAMDB, and the DB2 plan are referred to as IE, CE and DB2, respectively. For both IE and CE, the first execution of a plan is always somewhat slower than subsequent executions because of the JIT overhead. We also calculate a speedup (Sp) of CE over IE as IE/CE. The focus of this experimental study is on the comparison between IE and CE, and DB2 is used mainly as reference to make sure the JAMDB implementation is reasonable.

5.1 Sythethsized Workload Results

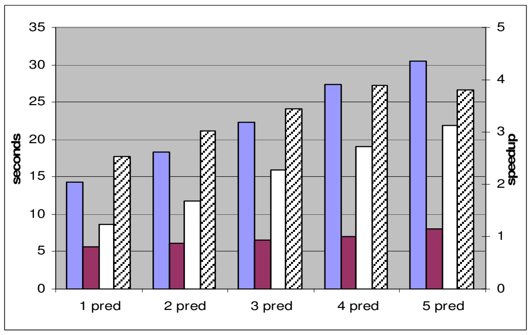
To better understand the performance tradeoffs, we start by testing some synthesized workloads. We designed five test sets, each testing four or five queries. Unless mentioned otherwise, the DB2 optimizer picks the same plan as JAMDB. In fact, for the first four test sets, both JAMDB and DB2 make a full table scan of lineitem table. All query templates and results are presented in Figure 5. We use only the initials for each table in query templates.

```
Test Set 1:
select count(*) from L
where  l_linestatus in ('1','2','3','4','5',
                        '6','7','8','9')
```

```
Test Set 2:
select count(*) from L
where l_shipdate <= '1978-12-01'or  l_commitdate <= '1978-12-01'
 or  l_receiptdate <= '1978-12-01' or  l_tax <= -1   or  l_discount <= -1
```



(a) increasing # of constants in a list (plans all tscan)

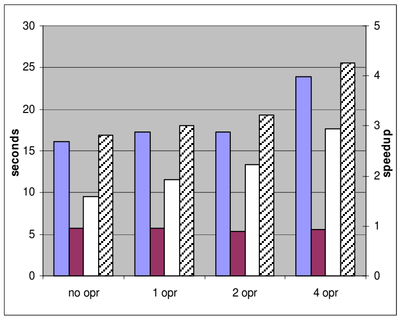


(b) increasing # of predicates (plans all tscan)

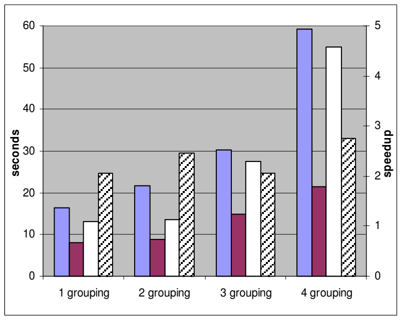
```
Test Set 3:
select sum(l_discount), sum(1 - l_discount),
        sum(l_extendedprice*(1-l_discount)),
        sum(l_extendedprice*(1-l_discount)
              *(1+l_tax))
from L
```

```
Test Set 4:
select  count(*),
          l_returnflag,l_linestatus,
          l_shipinstruct,l_shipmode
from L group by l_returnflag,l_linestatus,
                    l_shipinstruct,l_shipmode
```

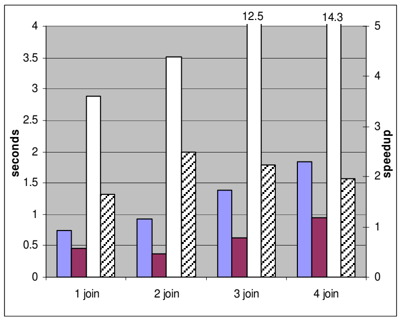
```
Test Set 5:
select  sum(l_extendedprice),sum(s_acctbal),
          sum(o_totalprice),sum(c_acctbal)
 from L,P,S,O,C where p_container='warp case' and
 l_partkey=p_partkey and l_suppkey=s_suppkey and
 l_orderkey=o_orderkey and o_custkey=c_custkey
```



(c) increasing expression complexity (plans all tscan)



(d) increasing # of grouping (plans all tscan)



(e) increasing # of joins

Figure 5 Synthesized Workload Results

In this test, we only show the results on AIX. Each testing point in the graphs has four bars representing IE, CE, DB2, and Sp, from left to right. The first three bars use the primary y-axis (where the unit is seconds) on the left, whereas the fourth bar uses the secondary y-axis (where the unit is times) on the right.

The first two test sets are designed to understand the overhead associated with evaluating Boolean predicates. The query template for Test Set 1 has a single "in" predicate. We run 5 queries, each using the first 1, 3, 5, 7, and all 9 constants in the template. We deliberately choose constants not present in l_linestatus. That way, all constants have to be checked by the predicate and there is no overhead in producing output (since it's empty). The results are shown in Figure 5(a). With each additional pair of constants, the time increase in IE is much larger than that of CE. In both IE and CE, column fetching is done only once for every tuple, independent of the number of constants, and thus query cost shifts to Boolean comparison cost with more and more constants. For each comparison, IE requires a function call. Such overhead is quite significant, given the small amount of work done inside that function. The speedup of CE over IE is about a factor of 3, which matches the function call overhead we measured earlier in Table 1 in Section 4.3. The trend for DB2 is very similar to that of IE, since both are interpreted engines. IE takes somewhat more time than DB2. This is likely because DB2 makes direct function calls while IE makes virtual function calls with more overhead. To help further understand where the speedup comes from, we collected hardware event counts of different levels of memory accesses and branch miss predictions (identified as affecting database operations the most in [1]) for IE and CE on AIX, using a monitoring tool called pmcount. Figure 6 compares four different events between IE and CE for test set 1. The event names for each pair of bars are given by the legend from left to right. As we can see, on average, IE introduces more than twice as many accesses as CE to all three levels of memory (note the log scale on the y-axis). Passing parameters through function calls prevents registers from being used, and thus increases the amount of cache and memory accesses. IE has significantly more branch miss predictions than CE. In this example, all Boolean tests will fail. In CE, all tests are inlined together. However, IE has to do the tests through virtual function calls, which require additional branch tests. This makes it harder for the processor to predict branching patterns.

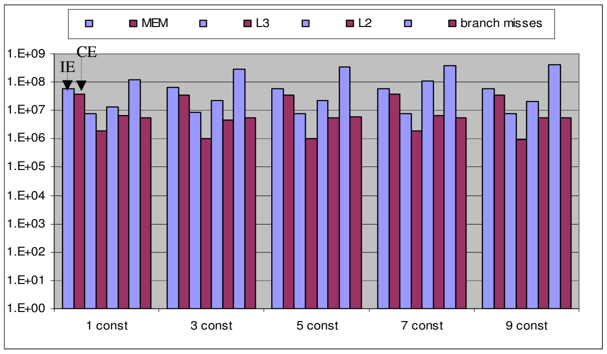


Figure 6 Hardware Events on AIX for Test Set 1

Test set 2 is very similar to test set 1. Each of the five queries applies an increasing number of Boolean predicates from the query template. The only difference is that each predicate now is on a different attribute, and therefore additional fetches are needed. Again, none of the predicates has any filtering power. As shown in Figure 5(b), the speedup here is slightly higher than that in test set 1. This is because each fetch in IE has the additional overhead of Java object casting. In CE, because of the specialization on a scan operator, such overhead is incurred only once per table.

We use test set 3 to evaluate arithmetic expressions. Each of the four queries in this test uses one of the expressions in the query template. All expressions are borrowed from TPC-H queries. To reduce the output overhead, we compute a scalar aggregate on each expression. Figure 5(c) shows the results, and each query is designated by the total number of binary arithmetic operations used in its expression. From the simplest expression to the most complex one, IE time increases by more than 50% while DB2 time doubles. In comparison, CE time remains almost unchanged. This is because, for each binary arithmetic operation, IE and DB2 have to make another function call. All operations for CE are inlined. What's more, the Java compiler now sees a larger block of code in CE and thus can perform optimizations such as subexpression elimination. It's very hard for the Java compiler to apply the same optimizations for IE, because the pieces of the binary expression code are in separate functions. The speedup of CE over IE increases with more complex expressions.

Next, test set 4 is designed to test the overhead of grouping. Queries in this test use 1 to 4 grouping columns from the query template, and compute a simple aggregate function. The results are shown in Figure 5(d). DB2 is not directly comparable to JAMDB, since it uses sort to implement grouping. Both IE and CE use the same hash function and are given the same number of hash buckets (more than the actual number of groups). The benefit of CE comes from two sources. First, CE inlines the comparisons on the grouping values while traversing through the hash chains. Second, CE refers to grouping columns directly, instead of going through an array structure. As a result, the speedup becomes larger with more grouping columns.

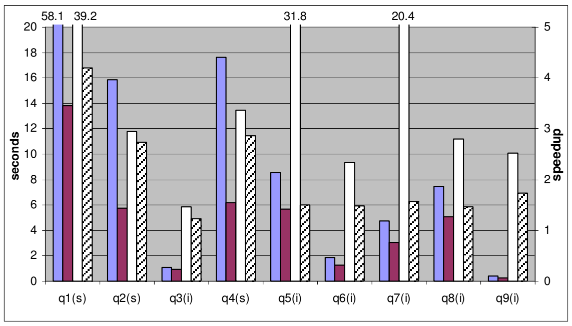
Finally, test set 5 tests the performance of joins. The first query applies a local predicate on p_container, joins table part with table lineitem, and then computes a scalar aggregate on l_extendedprice. Each subsequent query joins one more table, and computes an additional aggregate. The four queries are referred to as "1 join" to "4 join", respectively. The plan picked by JAMDB does an index scan on table part first, and follows each join index afterwards. The plans given by DB2 are the same for queries 1 join and 2 join, except that B+-tree indexes are used. For the other two queries, DB2 chooses a mix of hash joins and nested-loop joins. Figure 5(e) shows that the speedup of CE over IE is less than that in previous test sets. This is mainly due to two reasons. First, CE didn't specialize the index scan code and thus every tuple incurs the same casting overhead as IE. Second, we specialized IE code to fetch a column value from a referring tuple by following one or more fkLink, in a single function call, instead of one for each fkLink. Thus, the function call overhead is amortized in IE. Nevertheless, the average speedup of CE is still more than a factor of two.

5.2 TPC-H Queries

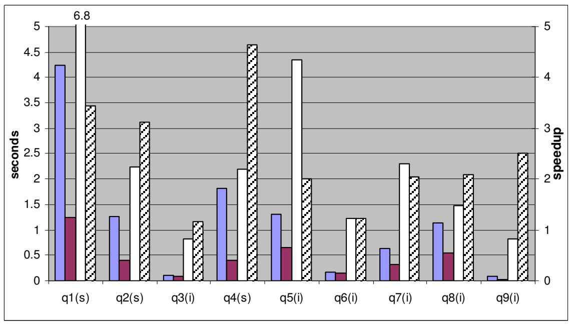
In this section, we present results on a real TPC-H workload. We tested 9 queries as listed in Figure 7, all of them are taken verbatim from TPC-H, except that q5 and q6 use fewer grouping columns to control the result size, and q9 ignores a subsequent arithmetic calculation because of JAMDB's limitation to a single query block (note that the subquery in q3 can be converted to a join). Those queries provide a variety of complexity on Boolean and arithmetic expressions, grouping fields, and joins (note that q7 and q9 have self joins). We order the queries by the number of tables referenced. To save space, we use "…" to represent the obvious natural join predicates. To account for differences in the groupby implementations between JAMDB and DB2, we further collect sorting cost in DB2 using its monitoring tool. The time spent in sorting is insignificant for all queries except for q1, in which about one third of the total time is spent on sorting.

The results on AIX and Windows are shown in Figures 7(a) and Figure 7(b), respectively. Inside the

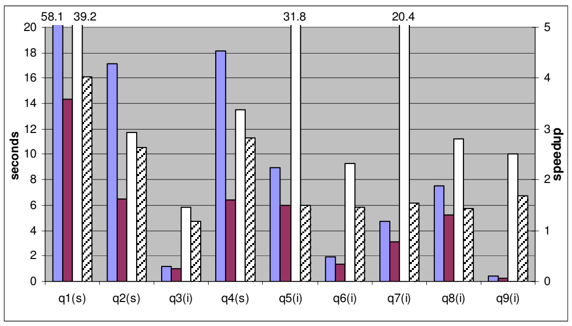
```
q1.
select l_returnflag, l_linestatus,
  sum(l_quantity),sum(l_extendedprice),
  sum(l_extendedprice*(1-l_discount)),
  sum(l_extendedprice*(1-l_discount)
        * (1 + l_tax)),
  avg(l_quantity),avg(l_extendedprice),
  avg(l_discount), count(*)
from L
where l_shipdate <= '1998-09-01'
group by l_returnflag, l_linestatus

q2.
select sum(l_extendedprice*l_discount)
from L where l_shipdate between
  '1994-01-01' and '1995-01-01'
 and l_discount between .059 and .061
 and l_quantity < 24
 group by o_orderpriority

q3.
select o_orderpriority, count(*)
from O where o_orderdate between
 '1993-07-01' and '1993-10-01'
and exists (select * from L
   where l_orderkey = o_orderkey
    and l_commitdate <= l_receiptdate)
group by o_orderpriority

q4.
select l_shipmode,
  sum(case
    when o_orderpriority = '1-urgent'
      or o_orderpriority = '2-high'
     then 1 else 0 end),
  sum(case
    when o_orderpriority <> '1-urgent'
      and o_orderpriority <> '2-high'
    then 1 else 0 end)
from O,L where …
and l_shipmode in ('MAIL', 'SHIP')
and l_commitdate <= l_receiptdate
and l_shipdate <= l_commitdate
and l_receiptdate between
  '1994-01-01' and '1995-01-01'
group by l_shipmode

q5.
select o_shippriority
  sum(l_extendedprice*(1-l_discount))
from C,O,L where …
and c_mktsegment = 'BUILDING'
and o_orderdate <= '1995-03-15'
and l_shipdate >= '1995-03-15'
group by o_shippriority
```



(a) AIX (SF 5)



(b) Windows (SF 1)



(c) AIX (SF5, nullable)

```
q6.
select n_name ,
  sum(l_extendedprice*(1-l_discount))
from C,O,L,N
where …
  and o_orderdate between
   '1993-10-01' and '1994-1-01'
  and l_returnflag = 'R'
group by n_name

q7.
select n1.n_name as s_nation,
  n2.n_name as c_nation,
  year (l_shipdate) as l_year,
  sum(l_extendedprice*(1-l_discount))
from S,L,O,C,N1,N2
where …
  and s_nationkey = n1.n_nationkey
  and c_nationkey = n2.n_nationkey
  and
   ((n1.n_name = 'ARGENTINA'
     and n2.n_name = 'BRAZIL')
    or
    (n1.n_name = 'BRAZIL'
     and n2.n_name = 'ARGENTINA'))
 and l_shipdate between
   '1995-01-01' and '1996-12-31'
group by s_nation, c_nation,l_year

q8.
select n_name,
  sum(l_extendedprice*(1-l_discount))
from C,O,L,S,N,R
where …
and c_nationkey = s_nationkey
and s_nationkey = n_nationkey
and r_name = 'AMERICA'
and o_orderdate between
  '1994-01-01' and '1995-01-01'
group by n_name

q9.
select year(o_orderdate) as o_year,
   n2.n_name,
   sum(l_extendedprice*(1-l_discount))
from  P,S,L,O,C,N1,N2,R
where …
and c_nationkey=n1.n_nationkey
and n1.n_regionkey=r_regionkey
and r_name='AMERICA'
and s_nationkey=n2.n_nationkey
and o_orderdate between
  '1995-01-01' and '1996-12-31'
and p_type = 'economy anodized steel'
group by o_year, n_name
```

Figure 7 TPC-H Query Results

bracket next to each query name, we use 's' and 'i' to represent whether JAMDB picks a scan or an index plan. The first observation is that on both platforms, CE improves IE more on the three scan plans (q1, q2 and q4), mainly because CE can save the casting overhead when fetching column values. Also, those 3 queries use more complex expressions and/or Boolean predicates. DB2 chooses the same plan for those three queries, except that q4 uses B-tree indexes in the nested-loop join.

Both IE and CE use the same index plans for all the other queries. The local index column in the outermost table is underlined in each query text (q7 normalizes disjuncts to conjuncts first). DB2's plans are mostly different from JAMDB and are used only as references. On both platforms, the speedup of CE over IE is less than that for the scan plans. Again, this is mainly because of our decision not to specialize the index iterator code. Nevertheless, the speedup is still significant and IE runs 50% longer than CE, on average. Among those queries, q3 and q9 have the smallest and the largest speedup, respectively. Compared with q3, q9 has more complexity in the aggregate expressions, and more grouping columns. Thus, more overhead can be reduced in q9 through code specialization in CE.

We measured the counts of the same four hardware events on AIX for the 9 queries and show the results in Figure 8. For queries accessing a large amount of data (q1-q4), IE incurs significantly more memory accesses. For the rest of the queries, much of the data needed is already in L3 or even L2 cache (p650 has a 32MB L3 cache and a 1.5MB L2 on-chip cache). Therefore, although the number of memory accesses is almost the same between IE and CE, IE introduces many more accesses in either L3 or L2 cache. Also, IE typically has at least twice as many branch miss predictions as CE.

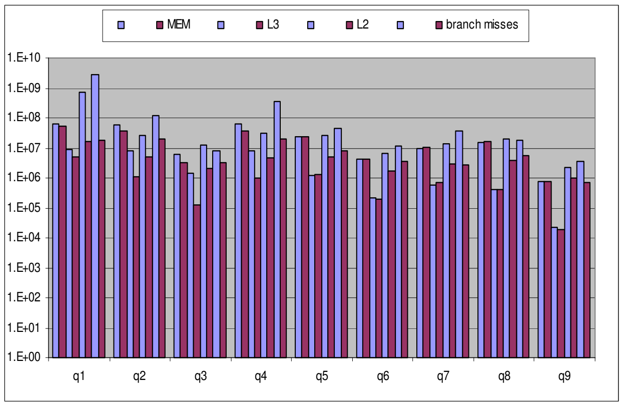


Figure 8 Hardware Events on AIX for TPC-H

SQL uses three-valued logic, which affects computations on nullable columns. For instance, a Boolean expression on a nullable column needs to additionally test if a column value is null or not. Similarly, an aggregate function needs an extra test in order to ignore all null values. In order to see the impact of this, we test the same 9 queries, assuming all columns are nullable (even though none of the null tests succeeds, since there are no null values in the underlying data). In IE, the computation is handled by separate null-supporting classes. The dynamic code generator adds additional tests in the generated code, depending on column nullability. We show the results on AIX in Figure 7(c). Compared with Figure 7(a), the speedup with null tests is slightly less (we did not test DB2 performance with all columns nullable). This is because null tests increase the amount of "real" work while the amount of overhead that CE saves remains the same. Nevertheless, such an impact is very small. A JIT compiler can, at runtime, figure out that those null tests often fail (likely true in real applications) and inline the false branch within the "if" statement when generating the machine code (note that we didn't do such an optimization explicitly in our generated Java code). Therefore, such null tests do not incur a branching cost, and can be done in one cycle. In JAMDB, we did not check data overflow or underflow for aggregate functions. However, since both are infrequent events, they can also be well optimized by JIT, and their impact to speedup is likely comparable to null testing.

Summary: We experimentally evaluated the difference between an interpreted engine and a compiled engine in JAMDB. For TPC-H queries, a compiled plan on average runs about twice as fast as an interpreted plan because of code specialization. Although not shown, tests using Sun's JVM on Windows show similar results, where the elapsed times are a little bit longer and the speedups are slightly larger. IE also runs much faster than DB2. Some of the reasons are because of the difference in the underlying data representation and the indexing structure, and that JAMDB does not have overhead due to latching and error handling in DB2 (such overhead is less than 5%, as measured by the monitoring tool tprof on AIX). Nevertheless, when plans are comparable, we observe that DB2 incurs a similar amount of overhead as IE, and the generated Java code in CE is more efficient than the static C/C++ implementation in DB2.

6 Discussion and Future Work

JAMDB did not explore seriously persistence and locking issues for a pure Java DBMS. In this section, we discuss some of the issues relating to logging, recovery and space management in this context. When JAMDB is used as a highly available multi-user server, it should be on par with disk based industrial strength systems with respect to recovery. The recovery requirements are much lighter for smaller databases with less availability requirements. Cache databases often fit this model. Cache databases are popular for use at the edge of the network or close to applications in the mid-tier [6]. Some of these databases often are reloaded from the server in case of a failure. In other cases, there is enough time to recover them from a full backup followed by forward recovery. However, in the context of highly available servers with larger databases, further exploration is required.

If each tuple is a Java object and such Java objects are stored in a large heap, then backing the in-memory version of the database to disk would be non-trivial if the size of the heap is large. We wouldn't want to interrupt the execution of transactions to take the time to write back atomically the whole heap. Doing the writes at a smaller granularity and reducing the number of such writes by keeping track of modified granules, would require more a structured way of using the heap (space allocation, synchronization of modifications via latches, etc.). Without a buffer manager, as in a traditional DBMS, we wouldn't know which objects are "dirty" and hence need to be written back to disk. Without the notion of a page, it is harder to deal with issues like detecting partial writing of data to disk, more economically using state identifiers like log sequence numbers (LSNs) for recovery purposes, as in the ARIES method [19], and tracking conveniently information about when an object was dirtied, etc.

Some of the approaches adopted in object-oriented database systems such as ObjectStore [34] (which currently supports persistent Java objects), Versant [37] and Ontos for supporting recovery would be applicable to this context, although the use of Java rather than C++ as the implementation language would introduce some additional complications. Recovery approaches adopted by the more traditional main-memory DBMS proposals/prototypes and recoverable VM proposals [27] would be applicable in our context. Persistent Java stores are available in some prototypes [3].

Even if main memory were to be non-volatile, we would still need to do logging and writing the memory contents to disk to handle failures of such a memory or to deal with catastrophes in which the whole installation is damaged. At least in-memory undo logging would be needed to support transaction rollback.

Currently, CE is manually written from scratch and the code is quite redundant (lots of print statements). We plan to rewrite CE using Java Emitter Templates (JET [28]), which is a powerful tool for generating source code through templates. A more severe problem is that CE code is harder to write (and to maintain) than IE. What one writes in CE is not really what one wants to run, but something that generates what one wants to run. Such a level of indirection often causes confusion when writing a CE. Further research is needed to simplify such a task.

Compared with IE, CE trades code size for performance. In general, this should not be a problem because code size for queries is typically much smaller than data. New code also has to be loaded into the instruction cache for CE. Nevertheless, the footprint of an IE is often larger than the instruction cache. Depending on the workload, instruction loads may also be needed. We close this section by noting that our approach can also be used in Microsoft's C# and .NET platform.

7 Conclusion

In this paper, we described a compiled query execution approach that exploits a JVM. Compared with a conventional SVM, our approach reduces runtime overhead and better exploits dynamic, low-level optimization by a JVM. Our experimental study on JAMDB validates that a compiled engine runs significantly faster than an interpreted engine in Java, although both can benefit from JIT optimizations. A compiled Java engine also runs faster than a C-based SVM with comparable plans, and the gap is likely to widen in the future with enhancements from the heavily invested JVM.

References

1 Anastassia Ailamaki, et al.: DBMSs on a Modern Processor: Where Does Time Go? VLDB 1999

2 Anastassia Ailamaki, et al.: Weaving Relations for Cache Performance. VLDB 2001: 169-180

3 Malcolm P. Atkinson, et al.: An Orthogonally Persistent Java. SIGMOD Record 25(4), 1996

4 Morton M. Astrahan, et al.: System R: A Relational Data Base Management System. IEEE Computer 12(5): 42-48 (1979)

5 P. Boncz, et al.: Database Architecture Optimized for the New Bottleneck: Memory Access. VLDB 1999

6 Christof Bornhövd, et al.: DBCache: Middle-tier Database Caching for Highly Scalable e-Business Architectures. SIGMOD Conference 2003: 662

7 Peter A. Boncz, et al.: MonetDB/X100: Hyper-Pipelining Query Execution. CIDR 2005: 225-237

8 C. Consel, et al.: A General Approach for Run-Time Specialization and its Application to C. POPL 1996

9 Rick Greer: Daytona And The Fourth-Generation Language Cymbal. SIGMOD Conference 1999

10 Goetz Graefe: Volcano - An Extensible and Parallel Query Evaluation System. TKDE 6(1), 1994

11 Stavros Harizopoulos, et al.: STEPS towards Cache-resident Transaction Processing. VLDB 2004

12 Joseph M. Hellerstein: Practical Predicate Placement. SIGMOD Conference 1994: 325-335

13 http://www.idiom.com/%7Ezilla/Computer/javaCbenchmark.html

14 http://incubator.apache.org/derby/

15 H. V. Jagadish, et al.: Dalí: A High Performance Main Memory Storage Manager. VLDB 1994: 48-59

16 http://jakarta.apache.org/bcel/

17 http://java.sun.com/products/hotspot/

18 Stefan Manegold, et al.: What Happens During a Join? Dissecting CPU and Memory Optimization Effects. VLDB 2000: 339-350

19 C. Mohan, et al.: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM Trans. Database Syst. 17(1): 94-162 (1992)

20 http://monetdb.cwi.nl/

21 Sriram Padmanabhan, et al.: Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. ICDE 2001

22 Jun Rao, et al.: Cache Conscious Indexing for Decision-Support in Main Memory. VLDB 1999

23 Jun Rao, et al.: Making B^+-Trees Cache Conscious in Main Memory. SIGMOD Conference 2000: 475-486

24 Kenneth Ross: Selection conditions in main memory. ACM Trans. Database Syst. 29: 132-161 (2004)

25 E. Schanzer, Performance Considerations for Run-Time Technologies in the .NET Framework, Microsoft Developer Network article.

26 Mehul Shah, et al.: Java Support for Data-Intensive Systems: Experiences Building the Telegraph Dataflow System. SIGMOD Record 30(4): (2001)

27 Alfred Z. Spector, et al.: The Camelot Project. IEEE Database Eng. Bull. 9(3): 23-34 (1986)

28 http://eclipse.org/emf/docs.php?doc=tutorials/jet1/jet_tutorial1.html

29 http://www.eclipse.org/

30 http://haifa.il.ibm.com/projects/systems/cot/fdpr/

31 http://www.ibm.com/software/data/cloudscape/

32 http://www.ibm.com/software/ db2/

33 http://www.jython.org/

34 http://www.objectstore.com/

35 http://www.oracle.com/timesten/

36 http://www.tpc.org/

37 http://www.versant.com/

38 http://xml.apache.org/xalan-j/xsltc_usage.html

39 J Zhou, et al.:Buffering Database Operations for Enhanced Instruction Cache Performance. Sigmod04