Compilation in the Microsoft SQL Server Hekaton Engine

Craig Freedman
craigfr@microsoft.com

Erik Ismert
eriki@microsoft.com

Per-Ake Larson
palarson@microsoft.com

Abstract

Hekaton is a new database engine optimized for memory resident data and OLTP workloads that is fully integrated into Microsoft SQL Server. A key innovation that enables high performance in Hekaton is compilation of SQL stored procedures into machine code.

1 Introduction

SQL Server and other major database management systems were designed assuming that main memory is expensive and data resides on disk. This assumption is no longer valid; over the last 30 years memory prices have dropped by a factor of 10 every 5 years. Today, one can buy a server with 32 cores and 1TB of memory for about $50K and both core counts and memory sizes are still increasing. The majority of OLTP databases fit entirely in 1TB and even the largest OLTP databases can keep the active working set in memory.

Recognizing this trend SQL Server several years ago began building a database engine optimized for large main memories and many-core CPUs. The new engine, code named Hekaton [2][3], is targeted for OLTP workloads.

Several main memory database systems already exist, both commercial systems [4][5][6][7][8] and research prototypes [9][10][11][12]. However, Hekaton has a number of features that sets it apart from the competition.

Most importantly, the Hekaton engine is integrated into SQL Server; it is not a separate DBMS. To take advantage of Hekaton, all a user has to do is declare one or more tables in a database memory optimized. This approach offers customers major benefits compared with a separate main-memory DBMS. First, customers avoid the hassle and expense of another DBMS. Second, only the most performance-critical tables need to be in main memory; other tables can be left unchanged. Third (and the focus of this article), stored procedures accessing only Hekaton tables can be compiled into native machine code for further performance gains. Fourth, conversion can be done gradually, one table and one stored procedure at a time.

Memory optimized tables are managed by Hekaton and stored entirely in main memory. Hekaton tables can be queried and updated using T-SQL in the same way as regular SQL Server tables. A query can reference both Hekaton tables and regular tables and a single transaction can update both types of tables. Furthermore, a T-SQL stored procedure that references only Hekaton tables can be compiled into native machine code. This is by far the fastest way to query and modify data in Hekaton tables and is essential to achieving our end-to-end performance goals for Hekaton.

The rest of the article is organized as follows. Section 2 outlines the high-level considerations and principles behind the design of Hekaton. Section 3 describes how stored procedures and table definitions are compiled into native code. Section 4 provides some experimental results.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Terminology. We will use the terms Hekaton table and Hekaton index to refer to tables and indexes stored in main memory and managed by Hekaton. Tables and indexes managed by the traditional SQL Server engine will be called regular tables and regular indexes. Stored procedures that have been compiled to native machine code will simply be called compiled stored procedures and traditional non-compiled stored procedures will be called interpreted stored procedures.

2 Design Considerations

Our goal at the outset of the Hekaton project was to achieve a 10-100X throughput improvement for OLTP workloads. An analysis done early on in the project drove home the fact that a 10-100X throughput improvement cannot be achieved by optimizing existing SQL Server mechanisms. Throughput can be increased in three ways: improving scalability, improving CPI (cycles per instruction), and reducing the number of instructions executed per request. The analysis showed that, even under highly optimistic assumptions, improving scalability and CPI can produce only a 3-4X improvement.

The only real hope is to reduce the number of instructions executed but the reduction needs to be dramatic. To go 10X faster, the engine must execute 90% fewer instructions and yet still get the work done. To go 100X faster, it must execute 99% fewer instructions. This level of improvement is not feasible by optimizing existing storage and execution mechanisms. Reaching the 10-100X goal requires a much more efficient way to store and process data.

So to achieve 10-100X higher throughput, the engine must execute drastically fewer instructions per transaction, achieve a low CPI, and have no bottlenecks that limit scalability. This led us to three architectural principles that guided the design.

2.1 Optimize indexes for main memory

Current mainstream database systems use disk-oriented storage structures where records are stored on disk pages that are brought into memory as needed. This requires a complex buffer pool where a page must be protected by latching before it can be accessed. A simple key lookup in a B-tree index may require thousands of instructions even when all pages are in memory.

Hekaton indexes are designed and optimized for memory-resident data. Durability is ensured by logging and checkpointing records to external storage; index operations are not logged. During recovery Hekaton tables and their indexes are rebuilt entirely from the latest checkpoint and logs.

2.2 Eliminate latches and locks

With the growing prevalence of machines with 100's of CPU cores, achieving good scaling is critical for high throughput. Scalability suffers when the systems has shared memory locations that are updated at high rate such as latches and spinlocks and highly contended resources such as the lock manager, the tail of the transaction log, or the last page of a B-tree index [13][14].

All of Hekaton's internal data structures, for example, memory allocators, hash and range indexes, and the transaction map, are entirely latch-free (lock-free). There are no latches or spinlocks on any performance-critical paths in the system. Hekaton uses a new optimistic multiversion concurrency control algorithm to provide transaction isolation semantics; there are no locks and no lock table [15]. The combination of optimistic concurrency control, multiversioning and latch-free data structures results in a system where threads execute without stalling or waiting.

2.3 Compile requests to native code

SQL Server uses interpreter based execution mechanisms in the same ways as most traditional DBMSs. This provides great flexibility but at a high cost: even a simple transaction performing a few lookups may require several hundred thousand instructions.

Hekaton maximizes run time performance by converting statements and stored procedures written in T-SQL into customized, highly efficient native machine code. The generated code contains exactly what is needed to execute the request, nothing more. As many decisions as possible are made at compile time to reduce runtime overhead. For example, all data types are known at compile time allowing the generation of efficient code.

The importance of compiling stored procedures to native code is even greater when the instruction path length improvements and performance improvements enabled by the first two architectural principles are taken into account. Once other components of the system are dramatically sped up, those components that remain unimproved increasingly dominate the overall performance of the system.

3 Native Compilation in Hekaton

As noted earlier, a key architectural principle of Hekaton is to perform as much computation at compile time as possible. Hekaton maximizes run time performance by converting SQL statements and stored procedures into highly customized native code. Database systems traditionally use interpreter based execution mechanisms that perform many run time checks during the execution of even simple statements. For example, the Volcano iterator model [1] uses a relatively small number of query operators to execute any query. Each iterator by definition must be able to handle a wide variety of scenarios and cannot be customized to any one case.

Our primary goal is to support efficient execution of compile-once-and-execute-many-times workloads as opposed to optimizing the execution of ad hoc queries. We also aim for a high level of language compatibility to ease the migration of existing SQL Server applications to Hekaton tables and compiled stored procedures. Consequently, we chose to leverage and reuse technology wherever suitable. We reuse much of the SQL Server T-SQL compilation stack including the metadata, parser, name resolution, type derivation, and query optimizer. This tight integration helps achieve syntactic and semantic equivalence with the existing SQL Server T-SQL language. The output of the Hekaton compiler is C code and we leverage Microsoft's Visual C/C++ compiler to convert the C code into machine code.

While it was not a goal to optimize ad hoc queries, we do want to preserve the ad hoc feel of the SQL language. Thus, a table or stored procedure is available for use immediately after it has been created. To create a Hekaton table or a compiled stored procedure, the user merely needs to add some additional syntax to the CREATE TABLE or CREATE PROCEDURE statement. Code generation is completely transparent to the user.

Figure 1 illustrates the overall architecture of the Hekaton compiler. There are two main points where we invoke the compiler: during creation of a Hekaton table and during creation of a compiled stored procedure.

As noted above, we begin by reusing the existing SQL Server compilation stack. We convert the output of this process into a data structure called the mixed abstract tree or MAT. This data structure is a rich abstract syntax tree capable of representing metadata, imperative logic, expressions, and query plans. We then transform the MAT into a second data structure called the pure imperative tree or PIT. The PIT is a much "simpler" data structure that can be easily converted to C code (or theoretically directly into the intermediate representation for a compiler backend such as Phoenix [17] or LLVM [16]). We discuss the details of the MAT to PIT transformation further in Section 3.2. Once we have C code, we invoke the Visual C/C++ compiler and linker to produce a DLL. At this point it is just a matter of using the OS loader to bring the newly generated code into the SQL Server address space where it can be executed.

3.1 Schema Compilation

It may not be obvious why table creation requires code generation. In fact, there are two reasons why table creation requires code generation.

The first reason is that the Hekaton storage engine treats records as opaque objects. It has no knowledge of the internal content or format of records and cannot directly access or process the data in records. The Hekaton compiler provides the engine with customized callback functions for each table. These functions perform tasks such as computing a hash function on a key or record, comparing two records, and serializing a record into a log buffer. Since these functions are compiled into native code, index operations such as inserts and searches are extremely efficient.

The second reason is that SQL Server's interpreted query execution engine can be leveraged to access Hekaton tables in queries that are not part of a compiled stored procedure. We refer to this method of accessing Hekaton tables as interop. While interop leverages the interpreted query execution engine code and operators, it does require some mechanism to crack Hekaton records and extract column values. When a new table is created, the Hekaton compiler determines the record layout and saves information about this record layout for use by the interop code. We discuss interop further in Section 3.4.

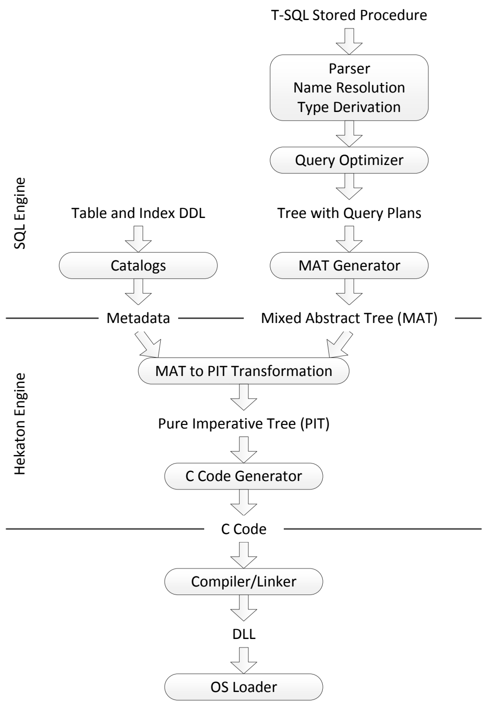


Figure 1: Architecture of the Hekaton compiler.

3.2 Stored Procedure Compilation

There are numerous challenging problems that we had to address to translate T-SQL stored procedures into C code. Perhaps the most obvious challenge is the transformation of query plans into C code and we will discuss our approach to this problem momentarily. There are, however, many other noteworthy complications. For example, the T-SQL and C type systems and expression semantics are very different. T-SQL includes many data types such as date/time types and fixed precision numeric types that have no corresponding C data types. In addition, T-SQL supports NULLs while C does not. 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.

These complexities were a major factor in our decision to introduce the intermediate step of converting the MAT into the PIT rather than directly generating C code. The PIT is a data structure that can be easily manipulated, transformed, and even generated out of order in memory. It is much more challenging to work directly with C code in text form.

The transformation of query plans into C code warrants further discussion. To aid in this discussion, consider the simple T-SQL example in Figure 2. This procedure retrieves a customer name, address, and phone number given a customer id. The procedure declaration includes some additional syntax; we will explain below why this syntax is required.

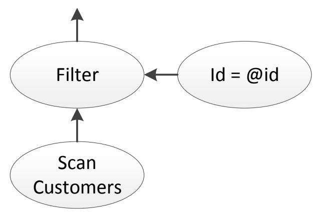
As with many query execution engines, we begin with a query plan which is constructed out of operators such as scans, joins, and aggregations. Figure 3(a) illustrates one possible plan for executing our sample query. For this example, we are naively assuming that the DBA has not created an index on Customer.Id and that the predicate is instead evaluated via a filter operator. In practice, we ordinarily would push the predicate down to the storage engine via a callback function. However, we use the filter operator to illustrate a more interesting outcome.

Each operator implements a common interface so that they can be composed into arbitrarily complex plans. In our case, this interface consists of GetFirst, GetNext, ReturnRow, and ReturnDone. However, unlike most query execution engines, we do not implement these interfaces using functions. Instead, we collapse an entire query plan into a single function using labels and gotos to implement and connect these interfaces. Figure 3(b) illustrates graphically how the operators for our example are interconnected. Each hollow circle represents a label while each arrow represents a goto statement. In many cases, we can directly link the code for the various operators bypassing intermediate operators entirely. The X's mark labels and gotos that have been optimized out in just such a fashion. In conventional implementations, these same scenarios would result in wasted instructions where one operator merely calls another without performing any useful work.

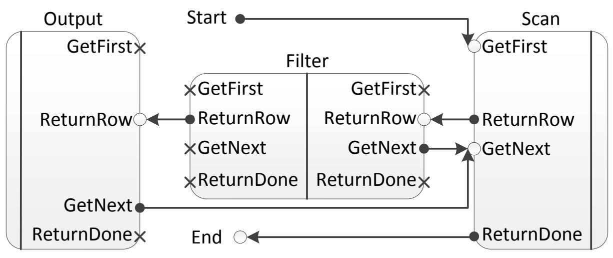
```
CREATE PROCEDURE SP_Example @id INT
WITH NATIVE_COMPILATION, SCHEMABINDING, EXECUTE AS OWNER
AS BEGIN ATOMIC WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT,
                      LANGUAGE = 'English')
   SELECT Name, Address, Phone FROM dbo.Customers WHERE Id = @id
END
```

Figure 2: Sample T-SQL procedure.

Execution of the code represented by Figure 3(b) begins by transferring control directly to the GetFirst entry point of the scan operator. Note already the difference as compared to traditional query processors which typically begin execution at the root of the plan and invoke repeated function calls merely to reach the leaf of the tree even when the intermediate operators have no work to do. Presuming the Customers table is not empty, the scan operator retrieves the first row and transfers control to the filter operator ReturnRow entry point. The filter operator evaluates the predicate and either transfers control back to the scan operator GetNext entry point if the current row does not qualify or to the output operator ReturnRow entry point if the row qualifies. The output operator adds the row to the output result set to be returned to the client and then transfers control back to the scan operator GetNext entry point again bypassing the filter operator. When the scan operator reaches the end of the table, execution terminates immediately. Again control bypasses any intermediate operators.



(a) Query plan



(b) Operator interconnects

Figure 3: Query plan and operator interconnects for sample T-SQL procedure.

This design is extremely flexible and can support any query operator including blocking (e.g., sort and group by aggregation) and non-blocking (e.g., nested loops join) operators. Our control flow mechanism is also flexible enough to handle operators such as merge join that alternate between multiple input streams. By keeping all of the generated code in a single function, we avoid costly argument passing between functions and expensive function calls. Although the resulting code is often challenging to read due in part to the large number of goto statements, it is important to keep in mind that our intent is not to produce code for human consumption. We rely on the compiler to generate efficient code. We have confirmed that the compiler indeed does so through inspection of the resulting assembly code.

Figure 4 gives a sample of the code produced for the sample procedure from Figure 2. We show only the code generated for the seek and filter operators with some minor editing for the sake of both brevity and clarity.

While the generated code ordinarily has no comments to protect against security risks and potential ”C injection attacks” (see Section 3.3), we do have the ability to add comments in internal builds for development and supportability purposes. The code in Figure 4 includes these comments for the sake of clarity. A careful analysis of this code sample will show that all of the gotos and labels are just as described above.

We compared this design to alternatives involving multiple functions and found that the single function design resulted in the fewest number of instructions executed as well as the smallest overall binary. This result was true even with function inlining. In fact, the use of gotos allows for code sharing within a single function. For example, an outer join needs to return two different types of rows: joined rows and NULL extended rows. Using functions and inlining with multiple outer joins, there is a risk of an exponential growth in code size [18]. Using gotos, the code always grows linearly with the number of operators.

```
    /*Seek*/
l_17:; /*seek.GetFirst*/
    hr = (HkCursorHashGetFirst(
        cur_15 /*[dbo].[Customers].[Customers_pk]*/ ,
        (context->Transaction),
        0, 0, 1,
        ((struct HkRow const**)(&rec1_16 /*[dbo].[Customers].[Customers_pk]*/ ))));
    if ((FAILED(hr)))
    {
        goto l_2 /*exit*/ ;
    }
l_20:; /*seek.1*/
    if ((hr == 0))
    {
        goto l_14 /*filter.child.ReturnRow*/ ;
    }
    else
    {
        goto l_12 /*query.ReturnDone*/ ;
    }
l_21:; /*seek.GetNext*/
    hr = (HkCursorHashGetNext(
        cur_15 /*[dbo].[Customers].[Customers_pk]*/ ,
        (context->ErrorObject),
        ((struct HkRow const**)(&rec1_16 /*[dbo].[Customers].[Customers_pk]*/ ))));
    if ((FAILED(hr)))
    {
        goto l_2 /*exit*/ ;
    }
    goto l_20 /*seek.1*/ ;
    /*Filter*/
l_14:; /*filter.child.ReturnRow*/
    result_22 = ((rec1_16 /*[dbo].[Customers].[Customers_pk]*/ ->hkc_1 /*[Id]*/ ) ==
        ((long)((valueArray[1 /*@id*/ ]).SignedIntData)));
    if (result_22)
    {
        goto l_13 /*output.child.ReturnRow*/ ;
    }
    else
    {
        goto l_21 /*seek.GetNext*/ ;
    }
```

Figure 4: Sample of generated code.

There are cases where it does not make sense to generate custom code. For example, the sort operator is best implemented using a generic sort implementation with a callback function to compare records. Some functions (e.g., non-trivial math functions) are either sufficiently complex or expensive that it makes sense to include them in a library and call them from the generated code.

3.3 Restrictions

With minor exceptions, compiled stored procedures look and feel just like any other T-SQL stored procedures. We support most of the T-SQL imperative surface area including parameter and variable declaration and assignment as well as control flow and error handling (IF, WHILE, RETURN, TRY/CATCH, and THROW). The query surface area is a bit more limited but we are expanding it rapidly. We support SELECT, INSERT, UPDATE, and DELETE. Queries currently can include filters, inner joins, sort and top sort, and basic scalar and group by aggregation.

In an effort to minimize the number of run time checks and operations that must be performed each time a compiled stored procedure is executed, we do impose some requirements.

First, unlike a conventional stored procedure which upon execution can inherit run time options from the user's environment (e.g., to control the behavior of NULLs, errors, etc.), compiled stored procedures support a very limited set of options and those few options that can be controlled must be set at compile time only. This policy both reduces the complexity of the code generator and improves the performance of the generated code by eliminating unnecessary run time checks.

Second, compiled stored procedures must execute in a security or user context that is predefined when the procedure is created rather than in the context of the user who executes the procedure. This requirement allows us to run all permission checks once at procedure creation time instead of once per execution.

Third, compiled stored procedures must be schema bound. This restriction means that once a procedure is created, any tables referenced by that procedure cannot be dropped without first dropping the procedure. This requirement avoids the need to acquire costly schema stability locks before executing the procedure.

Fourth, compiled stored procedures must execute in the context of a single transaction. This requirement is enforced through the use of the BEGIN ATOMIC statement (and the prohibition of explicit BEGIN, COMMIT, and ROLLBACK TRANSACTION statements) and ensures that a procedure does not have to block or context switch midway through to wait for commit.

Finally, as we are building a commercial database product, we must take security into account in all aspects of the design. We were particularly concerned about the possibility of a ”C injection attack” in which a malicious user might include C code in a T-SQL identifier (e.g., a table, column, procedure, or variable name) or string literal in an attempt to induce the code generator to copy this code into the generated code. Clearly, we cannot allow the execution of arbitrary C code by non-administrative users. To ensure that such an attack is not possible, we never include any user identifier names or data in the generated code even as comments and we convert string literals to a binary representation.

3.4 Query Interop

Compiled stored procedures do have limitations in the current implementation. The available query surface area is not yet complete and it is not possible to access regular tables from a compiled stored procedure. Recognizing these limitations, we implemented an additional mechanism that enables the interpreted query execution engine to access memory optimized tables. As noted in Section 3.1, we refer to this capability as interop. Interop enables several important scenarios including data import and export, ad hoc queries, support for query functionality not available in compiled stored procedures (including queries and transactions that access both regular and Hekaton tables).

4 Performance

We measured the benefits of native compilation through a set of simple experiments where we compare the number of instructions executed by the interpreted query execution engine with the number of instructions executed by an equivalent compiled stored procedure.

For the first experiment, we isolated and compared the cost of executing the simple predicate

```
  Item = ? and Manufacturer = ? and Price > ?
```

using both the interpreted expression evaluator and the equivalent native code. We measured the cost of this predicate when a) evaluated against data stored in a regular disk-based B-tree table, b) using interop to access data stored in a Hekaton table, and c) using a compiled stored procedure to access the same Hekaton table. We ran these experiments without a suitable index as our intent was to measure the cost of the predicate evaluation not the cost of an index lookup. Since short circuiting can impact the cost of evaluating the predicate, we also measured the best (first column does not match) and worst (all columns match) case costs. The results are shown in Table 1. We use the most expensive case (the interpreted expression evaluator against a regular table with a row with all columns matching) as the baseline and report all other results as a percentage of this baseline.

Not surprisingly the compiled code is much more efficient (up to 10x fewer instructions executed) than the interpreted code. This improvement reflects the benefits of generating code where the data types and operations to be performed are known at compile time.

For our second experiment, we ran the simple queries shown in Figure 5. The only difference between these queries is that the first outputs the results to the client while the second saves the result in a local variable. Because outputting to the client incurs relatively high overhead, the first query is considerably more expensive than

	Interpreted expression (regular table)	Interop (Hekaton table)	Compiled (Hekaton table)
Worst case (all columns match)	100%	47%	11%
Best case (first column does not match)	43%	16%	4%

Table 1: Relative instruction cost for evaluating a predicate

the second. We ran the same three tests as for the first experiment comparing the interpreted query execution engine against a regular table, interop against a Hekaton table, and a compiled stored procedure against a Hekaton table. For this experiment we created an index on the Item and Manufacturer columns to simulate a more realistic scenario. The results are shown in Table 2. Once again we use the most expensive case (the interpreted query execution engine against a regular table with the results output to the client) as the baseline.

	Interpreted expression (regular table)	Interop (Hekaton table)	Compiled (Hekaton table)
Output to client	100%	66%	19%
Output to variable	94%	61%	6%

Table 2: Relative instruction cost for evaluating a query.

Once again, we see that the compiled code is much more efficient (up to 15x fewer instructions executed) than the interpreted code. As in the first experiment, many of the gains come from compile time knowledge of the data types and operations. We are also able to eliminate many virtual function calls and conditional branches whose outcomes are known at compile time.

```
--  Q1: Output to client
SELECT CustId FROM Sales
WHERE Item = ?
  AND Manufacturer = ?
  AND Price > ?

--  Q2: Output to variable
SELECT @CustId = CustId FROM Sales
WHERE Item = ?
  AND Manufacturer = ?
  AND Price > ?
```

Figure 5: Sample queries used in experiments.

Finally, a comment regarding the cost of compilation. Hekaton's primary focus is on OLTP queries where we compile once and execute thousands or millions of times. Thus, compiler performance was not a primary focus of this project. Nonetheless, for the vast majority of compiled stored procedures, the impact of compilation is not noticeable to the user. Most compilations complete in under one second and in many cases the cost of existing steps such as query optimization remain a significant fraction of the total end-to-end procedure creation cost. Some extremely complex procedures take longer to compile with the bulk of the time spent in compiler optimization. While such scenarios are rare, in the event that compilation takes unacceptably long, disabling some or all compiler optimizations generally improves compilation costs to an acceptable level albeit at some loss of runtime performance.

5 Concluding Remarks

Hekaton is a new database engine targeted for OLTP workloads under development at Microsoft. It is optimized for large main memories and many-core processors. It is fully integrated into SQL Server, which allows customers to gradually convert their most performance-critical tables and applications to take advantage of the very substantial performance improvements offered by Hekaton.

Hekaton achieves its high performance and scalability by using very efficient latch-free data structures, multiversioning, a new optimistic concurrency control scheme, and by compiling T-SQL stored procedure into efficient machine code. As evidenced by our experiments, the Hekaton compiler reduces the instruction cost for executing common queries by an order of magnitude or more.

References

[1] Goetz Graefe: Encapsulation of Parallelism in the Volcano Query Processing System. ACM SIGMOD 1990: 102-111

[2] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, Mike Zwilling: Hekaton: SQL server's memory-optimized OLTP engine. ACM SIGMOD 2013: 1243-1254

[3] Per-Ake Larson, Mike Zwilling, Kevin Farlee: The Hekaton Memory-Optimized OLTP Engine. IEEE Data Eng. Bull. 36(2): 34-40 (2013)

[4] IBM SolidDB, http://www.ibm.com/software/data/soliddb

[5] Oracle TimesTen, http://www.oracle.com/technetwork/products/timesten/overview/index.html

[6] SAP In-Memory Computing, http://www.sap.com/solutions/technology/in-memorycomputing-platform/hana/overview/index.epx

[7] Sybase In-Memory Databases, http://www.sybase.com/manage/in-memory-databases

[8] VoltDB, http://voltdb.com

[9] Martin Grund, Jens Krger, Hasso Plattner, Alexander Zeier,Philippe Cudr-Mauroux, Samuel Madden: HYRISE - A Main Memory Hybrid Storage Engine. PVLDB 4(2): 105-116 (2010)

[10] Martin Grund, Philippe Cudr-Mauroux, Jens Krger, Samuel Madden, Hasso Plattner: An overview of HYRISE - a Main Memory Hybrid Storage Engine. IEEE Data Eng. Bull.35(1): 52-57 (2012)

[11] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, Daniel J. Abadi: H-store: a high-performance, distributed main memory transaction processing system. PVLDB 1(2): 1496-1499 (2008)

[12] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, Anastasia Ailamaki: Data-Oriented Transaction Execution. PVLDB 3(1): 928-939 (2010)

[13] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, Michael Stonebraker: OLTP through the looking glass, and what we found there. SIGMOD 2008: 981-992

[14] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, Babak Falsafi: Shore-MT: a scalable storage manager for the multicore era. EDBT 2009: 24-35

[15] Per-Ake Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, Mike Zwilling: High-Performance Concurrency Control Mechanisms for Main-Memory Databases. PVLDB 5(4): 298-309 (2011)

[16] The LLVM Compiler Infrastructure, http://llvm.org/

[17] Phoenix compiler framework, http://en.wikipedia.org/wiki/Phoenix_compiler_framework

[18] Thomas Neumann: Efficiently Compiling Efficient Query Plans for Modern Hardware. PVLDB 4(9): 539-550 (2011)