# Measurement-based Cost Estimation Method of a Join Operation for an In-Memory Database

Tsuyoshi Tanaka* and Hiroshi Ishikawa†

Faculty of System Design, Tokyo Metropolitan University, Tokyo, Japan
Email: *tanaka-tsuyoshi@ed.tmu.ac.jp and †ishikawa-hiroshi@tmu.ac.jp

***Abstract*—Non-volatile memory is applied not only to storage subsystems but also to main memory to improve performance and increase capacity. Some in-memory database systems use non-volatile main memory as a durable medium instead of using existing storage devices such as hard disk drives or solid state drives. For such in-memory database systems, the cost of memory access instead of I/O processing decreases, and the CPU cost increases relatively for cost calculation to select the most suitable access path for a database query. Therefore, a high-precision cost calculation method of query execution is required. In particular, when the database system cannot select a proper join method, the query execution time increases. Accordingly, we propose a database join operation cost model using statistics information measured by a performance monitor embedded in the CPU and evaluated the accuracy of estimating the change point of join methods. As a result, the proposed method can estimate more accurately than the existing method to within one significant figure. In conclusion, the in-memory database system using the proposed cost calculation method is able to select the best join method.**

***Keywords–Non-volatile memory; In-memory database systems; Query optimization; Query execution cost.***

## I. INTRODUCTION

Improving the performance and expanding the capacity of non-volatile memory (NVM) is made applicable to both high-speed disk drives and main memory units. Intel and Micron developed the NVM named 3D Xpoint memory [1] for such use. NVM is implemented as byte-addressable memory and is assigned as a part of the main memory space. An application programming interface (API) [2] [3] for accessing NVM is proposed to make the development of applications easier. Roughly speaking, the API provides two types of access methods to NVM from software. The first is “load/store type:“ it is the same method used to access conventional main memory from user applications. The other is “read/write type:“ this is the method used by existing I/O devices, such as hard disk drives (HDDs) or solid state drives (SSD) through operating system (OS) calls such as read/write functions. There are two types of implementations of in-memory databases through the application of NVM to main memory. The load/store type must be implemented using array structures or list structures on a main memory address area such as the durable media of the database (Figure 1(c)). The read/write type can be easily applied to the existing database management system (DBMS) because the database files stored on disk drives (Figure 1(a)) are moved to files on NVM defined by the API for NVM (Figure 1(b)). The performance when accessing the database using the former type is better than the latter type because the DBMS directly accesses the database without any I/O device emulation operation. However, operations of the database administration (e.g., system configuration, backup, etc.) do not have to be changed. That means that it is easy for the administrators to introduce the in-memory database system.

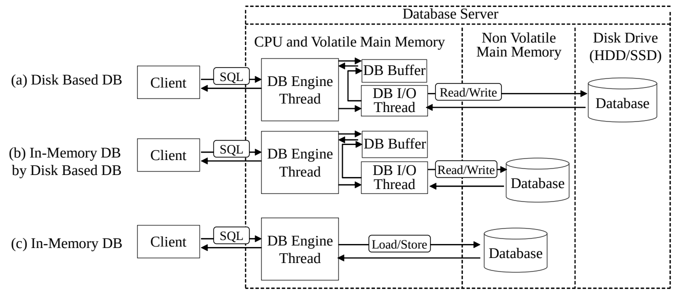


Figure 1. Disk-based database and in-memory database

The DBMS has a problem in preparation for executing a query. In general, the DBMS executes several steps before executing a query. First, the DBMS analyzes the query. Next, it creates multiple execution plans. Then, it estimates the query processing cost for each execution plan. Finally, it selects a minimum execution plan from a plurality of candidates. For example, when the DBMS joins two tables, such as the R table and S table shown in Figure 2(a), it generates the execution plan (Figure 2(b)) that minimizes the number of rows to be referenced. At this time, the execution time depends on which join method the DBMS selects. The DBMS estimates the cost of each join method by using statistical information from the database and chooses the join method with the minimum cost. In general, the cost of a join operation is a function of the ratio of the extracted records to all records. Hereafter, we refer to this ratio as the selectivity. In Figure 2, the selectivity is determined by the condition $x$ for the column R.C in Figure 2(c). In Figure 2(c), two cost functions cross at $\mathrm{X_{cross}}$. Join method 2 must be chosen from the left side of $\mathrm{X_{cross}}$ and join method 1 should be chosen from the right side of $\mathrm{X_{cross}}$. If the DBMS cannot estimate the selectivity $\mathrm{X_{cross}}$ accurately, it will choose the wrong join method.

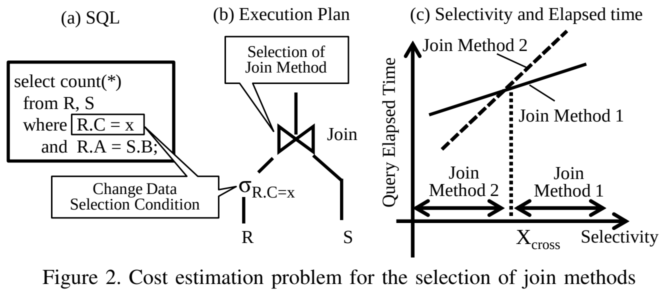


Figure 2. Cost estimation problem for the selection of join methods

On the other hand, the query execution cost ($cost$) is generally expressed as the sum of the Central Processing Unit (CPU) cost ($cpu\_cost$) and the I/O cost ($io\_cost$) [4] [5]. The CPU cost is the CPU time, and the I/O cost is the latency when accessing the disk drive:

$$cost = cpu\_cost + io\_cost \tag{1}$$

For example, the cost formula for MySQL is given below [6]. The cost of scanning a table R is given by

$$table\_scan\_cost(\mathrm{R}) = record(\mathrm{R}) \times CPR + page(\mathrm{R}) \times CPIO \tag{2}$$

where $record(\mathrm{R})$ is the number of records of table R, $CPR$ is the CPU cost per record, $page(\mathrm{R})$ is the number of pages of table R and $CPIO$ is the I/O cost per page stored record for DBMS access. When table R (inner table) and table S (outer table) are joined, the cost of a join operation is given by

$$table\_join\_cost(\mathrm{R,S}) = table\_scan\_cost(\mathrm{R}) + record(R) \times selectivity \times records\_per\_key(\mathrm{S}) \times (CPIO + CPR) \tag{3}$$

where $selectivity$ is the selectivity ratio given by the distribution of attributes, and the condition for selection such as a where-clause definition in SQL, and $records\_per\_key(\mathrm{S})$ is the number of join keys specified by table S's records. Here $CPR = 0.2$ and $CPIO = 1$ are the default defined values. However, this cost model is established under the condition that I/O performance is the bottleneck of the query execution time. A further improvement in disk performance increases the CPU cost relative to the I/O cost. When the I/O cost itself disappears ultimately in a native in-memory database (Figure 1(c)), it becomes necessary to more accurately predict the CPU cost.

To improve the accuracy of the CPU processing cost prediction, the estimation of CPU processing time must become more accurate than the conventional method mentioned above. In general, the CPU processing time can be predicted by the product of the number of executed instructions and the latency until the instruction is completed. To estimate the latency with high accuracy, it is necessary to consider the structure of the hardware, such as instruction execution parallelism, cache miss ratio, and memory hierarchy. These are problems that cannot be solved by the software algorithm alone.

In this study, we propose a method to improve the accuracy of CPU cost estimation of in-memory databases applied to existing DBMSs (Figure 1(b)). It is easy to apply our method to native in-memory databases (Figure 1(c)). Our contribution can be summarized as follows.

- First, we propose a method for modeling CPU cycles and estimating the join operation cost for a database. While considering the CPU pipeline architecture, we classify CPU cycles into three components: a pipeline stall cycle caused by instruction cache misses, a pipeline stall cycle caused by branch misprediction, and an access cycle of data caches or main memory. By using this classification, we propose a CPU cycle modeling method, which can express the total CPU execution time. In addition, to estimate the processing time of the join operation of a database, we decompose the pattern of the join processing into four parts and estimate the join operation cost by using a combination of these parts (Section II).
- Next, we analyze the behavior of measurement results of join operation by using a performance monitor embedded on the CPU and determine the cost estimation formulas (Section III).
- Finally, we verify the accuracy of the proposed CPU cost estimation formulas by comparing the actual CPU processing cycle and the conventional CPU cost estimation formula of MySQL (Section IV).

## II. Proposed CPU Cost Model

In this section, first, we analyze the CPU pipeline architecture and categorize pipeline events. We propose the CPU operation cycle estimation method, which can express whole CPU process cycles by considering the categorized events. Next, we categorize join operations of the DBMS and divide the join operation into several parts. We propose an estimation model based on a combination of these parts. Finally, we create the CPU cost formula for estimating each part of the join operations using statistics information measured by the performance monitor embedded in the CPU and assemble those join parts formulas into the complete CPU cost estimation formula.

### *A. Model of CPU Operation Time*

We chose the Intel Nehalem processor as a typical model of a CPU for application to the database server because all of the processors developed after Nehalem, namely Sandy Bridge, Haswell, and Skylake, are based on the pipeline architecture of Nehalem. Partial enhancements, such as additional cache for the micro-operations (uOPs), increased reorder buffer entries, and increased instruction execution units, were added to the successor CPUs of Nehalem.

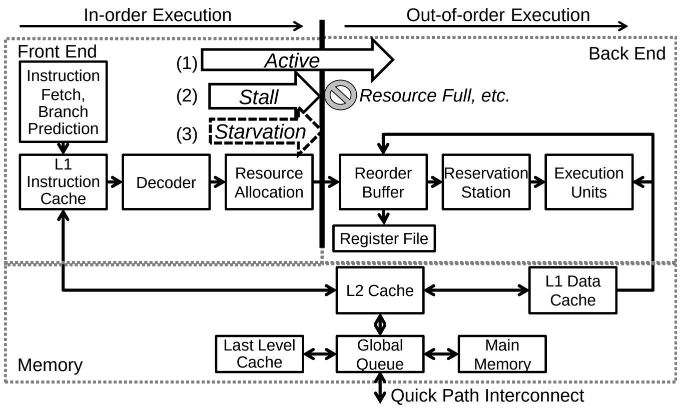


Figure 3. Focus point of the CPU pipeline

The pipeline is composed of a front-end and a back-end in Figure 3 [7]. The front-end fetches instructions from the L1 instruction cache (L1I) and decodes them into uOPs in-order. The term "in-order" means that a subsequent instruction cannot overtake preceding instructions in the pipeline. After decoding instructions, the front-end issues uOPs to the back-end. Conversely, the back-end executes the uOPs in execution units out-of-order. The back-end can execute the uOPs in a different order than issued by the front-end to improve the throughput of operating uOPs. An L1I miss causes the pipeline of the front-end to stall until the missing instruction is fetched from the lower level of cache or main memory. A branch prediction miss causes a dozen cycles of the instructions executed speculatively to be flashed, and the front-end cannot issue uOPs. In this paper, such a condition is referred to as an *instruction-starvation state* (Figure 3(3)). There are cases in which the uOP issued in the front-end is not executed because of the saturation of the reorder buffer or the reservation station in the back-end, or the data dependency with the preceding instructions. We refer to this state as a *stall state* in this paper (Figure 3(2)). In addition, we refer to the state in which the uOPs are issued without an *instruction-starvation state* or a *the stall state* as an *active state*.

A summary of the related notation of the CPU cost calculation to be used afterward is shown in Table I before creating the CPU cost calculation model.

TABLE I. NOTATION FOR THE CPU COST CALCULATION MODEL

| **Symbol** | **Description** |
|---|---|
| $I$ | Number of instructions to complete a query |
| $CPI0$ | Cycle per instruction (CPI) on the condition that all of instructions and data are stored in L1 cache |
| $Li$ | Level $i$ cache memory (Maximum value of “$i$“ varies depending on the CPU. In this paper, the maximum is “3“. L3 is represented by “last-level cache“ (LLC). |
| $M_{mem}$,$L_{mem}$ | Number of references of instructions and data to the main memory, main memory latency |
| $M_{memI}$,$M_{memD}$ | Number of references of instructions to the main memory, number of references of data to the main memory |
| $M_{Li}$,$L_{Li}$ | Number of references of instructions and data to L$i$ cache, L$i$ cache latency |
| $M_{LiI}$,$M_{LiD}$ | Number of references of instructions to L$i$ cache, the number of references of data to L$i$ cache |
| $BF_{mem}$,$BF_{Li}$ | Blocking factor of main memory, L$i$ cache reference |
| $BF_{memI}$,$BF_{LiI}$ | Blocking factor of instruction references to the main memory, L$i$ cache |
| $BF_{memD}$,$BF_{LiD}$ | Blocking factor of data references to the main memory, L$i$ cache |
| $BF_{MP}$ | Blocking factor when branch misprediction and instruction cache miss occur simultaneously |
| $H_{mem}$ | Ratio of the number of references to the main memory to the number of instructions ($H_{mem} = M_{mem}/I$) |
| $H_{Li}$ | Ratio of the number of references to L$i$ cache to the number of instructions ($H_{Li} = M_{Li}/I$) |
| $H_{LiI}$ | Ratio of instruction references to L$i$ cache to the number of instructions |
| $H_{LiD}$ | Ratio of data references to L$i$ cache to the number of instructions |
| $C_{Total}$ | Total CPU cycles to execute a query |
| $C_{Active}$,$C_{Stall}$, $C_{Starvation}$ | CPU cycles in *active state*, *stall state*, *starvation state* |
| $C_{ICacheMiss}$ | CPU cycles from the occurrence of L1I miss until the acquisition of an instruction from other cache or the main memory |
| $C_{DCacheAcc}$ | CPU cycles in *active state* |
| $M_{MP}$ | Number of branch mispredictions |
| $L_{MP}$ | Recovering latency from a branch misprediction |
| $C_{MP}$ | Total CPU cycles when recovering from branch mispredictions |
| $P$ | Selectivity of the outer table |
| $R_O$, $R_I$ | Number of outer table records, number of inner table records |
| $T_{NLJ}$, $T_{HJ}$ | Nested loop join (NLJ) execution time, hash join (HJ) execution time |
| $T_{build}$, $T_{probe}$ | Execution time of the HJ build phase, execution time of the HJ probe phase |
| $C_{NLJ\_Total}$ | Total CPU cycles of NLJ |
| $C_{NLJ\_ICacheMiss}$ | CPU cycles from the occurrence of L1I miss on executing NLJ until acquisition of an instruction from other cache or the main memory |
| $C_{NLJ\_MP}$ | Total CPU cycles when recovering from branch mispredictions on executing NLJ |
| $C_{NLJ\_DCacheAcc}$ | CPU cycles in *active state* on executing NLJ |
| $C_{Build\_Total}$, $C_{Probe\_Total}$ | Total CPU cycles of the build phase of HJ, probe phase of HJ |
| $C_{Build\_ICacheMiss}$, $C_{Probe\_ICacheMiss}$ | CPU cycles from the occurrence of L1I miss until the acquisition of an instruction from other cache or the main memory on executing the build phase of HJ, probe phase of HJ |
| $C_{Build\_MP}$, $C_{Probe\_MP}$ | Total CPU cycles of recovering from branch mispredictions on executing the build phase of HJ, probe phase of HJ |
| $C_{Build\_DCacheAcc}$, $C_{Probe\_DCacheAcc}$ | CPU cycles in data cache or main memory Access on executing the build phase of HJ, probe phase of HJ |
| $I_{Load}$ | Number of load instructions |
| $M_{LMMI}$, $M_{LMMD}$ | Number of instruction references to local main memory, number of data references to local main memory |
| $L_{LMM}$ | Latency of local main memory |
| $M_{LLLCI}$, $M_{LLLCD}$ | Number of instruction references to local LLC, number of data references to local LLC |
| $L_{LLLC}$ | Latency of local LLC |
| $M_{RLLCI}$, $M_{RLLCD}$ | Number of instruction references to remote LLC, number of data references to remote LLC |
| $L_{RLLC}$ | Latency of remote LLC |

In this paper, we focus on the boundary between the front-end and the back-end in the CPU pipeline (Figure 3) to model the overall operation of the CPU. The uOPs are issued from front-end to back-end and are stored in the buffers, namely the reorder buffer and reservation station. The buffers allow us to change the processing order of uOPs from in-order to out-of-order across the boundary. The CPU-embedded performance monitor can measure events such as the saturation of buffers, dequeues from buffers by the completion of uOPs, and the existence of uOPs to issue to back-end [7]. Any CPU cycle situation can be modeled by the performance monitor to analyze these events. Therefore, we propose measurement-based estimation of the query execution cost. The *active state* is estimated from the number of the events that the uOP is issued without delay in the back-end buffer. The buck-end buffer holds the uOPs until the execution of the uOPs is completed and the uOPs are deleted from the buffer. The *stall state* is estimated from the number of the events for which the buffer cannot receive uOPs. The *starvation state* is inferred from the event count in which there are no uOPs to be issued to the back-end buffer. The total CPU cycle is composed of the *active state* cycle, the *stall state* cycle and the *starvation state* cycle. Therefore, the following equation can be obtained:

$$C_{Total} = C_{Active} + C_{Stall} + C_{Starvation} \tag{4}$$

Cycle Per Instruction (CPI), which refers to the number of CPU clock cycles per instruction, is widely used as a metric for evaluating CPU processing efficiency [8]. CPI is calculated as the product of the number of references to the memory and the latency of the memory access. Latency is the delay time when fetching an instruction or data from memory. CPI is given by

$$CPI = CPI0 + \{ \sum_{i=2}^{last\_level} (H_{Li} \times L_{Li} \times BF_{Li}) + (H_{mem} \times L_{mem} \times BF_{mem}) \} \tag{5}$$

where last-level cache ( LLC) means the lowest cache in the cache memory hierarchy and the blocking factor [8] is a correction coefficient for concealing the latency by executing instructions in parallel. The second term on the right-hand side of (5) is the product of the number of memory references, the latency, and the blocking factor, i.e., the *stall state*. The product of the second term on the right-hand side of (5) and the number of instructions $I$ is the pipeline stall cycle ($C_{Stall}$):

$$C_{Stall} = \sum_{Li=L2}^{LLC} (M_{Li} \times L_{Li} \times BF_{Li}) + (M_{mem} \times L_{mem} \times BF_{mem}) \tag{6}$$

$$C_{Total} = CPI \times I = CPI0 \times I + C_{Stall} \tag{7}$$

From (5)–(7), we can show that $CPI0$ includes the *active state* and *starvation state*:

$$CPI0 \times I = C_{Active} + C_{Starvation} \tag{8}$$

The *starvation state* is mainly caused by instruction cache misses or branch mispredictions, and can be classified as the number of CPU cycles from the occurrence of one of these events until the acquisition of the next instruction to be executed:

$$C_{Starvation} = C_{ICacheMiss} + M_{MP} \times L_{MP} \times BF_{MP} \tag{9}$$

$$C_{ICacheMiss} = \sum_{Li=L2}^{LLC} (M_{LiI} \times L_{Li} \times BF_{LiI}) + (M_{memI} \times L_{mem} \times BF_{memI}) \quad (10)$$

Here $BF$ is a correction coefficient for considering that both branch misprediction and instruction cache miss occur simultaneously. $ICacheMiss$ is expressed as 10 by modifying 6 because operations after instruction cache misses and data cache misses are the same. Only terms relating to branch misprediction are defined:

$$C_{MP} = M_{MP} \times L_{MP} \times BF_{MP} \quad (11)$$

According to the previous research [9], the CPI of the decision support system benchmark is 1.5–2.5. In general, when the CPI is 1, this means that one instruction is completed in one cycle, so the instructions are executed sequentially in query execution. In addition, since the indices and tables of the database are usually implemented with list structures or tree structures, it is not until the stored data which the pointer refers to is read out that the next reference address becomes clear. Thus, it is difficult for the CPU to predict the destination of the next reference. In particular, the characteristics of such a memory reference in the list structure are applied to a benchmark program for measuring memory latency [10]. Therefore, *stall state* occurs because the operation of the stalled instruction waits for the preceding data reference processing to be completed. From the viewpoint of memory reference, the *active state* can be considered as an L1 data cache (L1D) reference, and the *stall state* can be considered as a reference to a cache level lower than L1 or a main memory reference. Therefore, CPU cycles in the *active state* and those in the *stall state* can be integrated as $C_{DCacheAcc}$ in

$$C_{DCacheAcc} = C_{Active} + C_{Stall} \quad (12)$$

$$C_{DCacheAcc} = \sum_{i=1}^{last_level} (M_{LiD} \times L_{Li} \times BF_{LiD}) + (M_{memD} \times L_{mem} \times BF_{memD}) \quad (13)$$

where (6)(13) use the same symbols for latency and the blocking factor for convenience, but the contents are different.

From the above discussion, the total number of CPU cycles is calculated using

$$C_{Total} = C_{DCacheAcc} + C_{ICacheMiss} + C_{MP} \quad (14)$$

In this paper, each term on the right-hand side of (14) uses statistical information obtained from actual measurements.

## *B. DBMS Operation Model*

DBMS queries perform operations including selection, projection, and join. Queries performing the join operation depend on the join method chosen by the DBMS's optimizer. The optimizer selects the join method to minimize the operation cost of the join operation. The cost depends on the selectivity of records defined by the clause of the SQL and the statistics of the attribute value of the database. Most DBMSs calculate the statistics during data loading to the database. This paper focuses on the cost estimation for the optimization of join operations. There are three basic joins: nested loop join (NLJ), hash join (HJ), and sort–merge join (SMJ).

NLJ searches records from the inner table every time it reads one record from the outer table. The generalized operation model of NLJ is shown in Figure 4. The process involves tracing multiple tables and indices from the point of view of memory access, which means repeatedly traversing linked lists. Therefore, NLJ can be regarded as searching between the outer table and the huge internal table created by tracing multiple tables in the same way as loop expansion by a compiler. Moreover, it is possible to calculate the cost of NLJ of multiple tables using the cost estimation function with two typical NLJs (Figure 4(a)), which is the function of the number of total records to be referenced in the multi-table join. NLJ and HJ are regarded as part of our proposed cost estimation method. In this paper, we do not examine SMJ because it is possible to apply the proposed method using the steps from the other join methods, specifically dividing parts into sorting and merging operations and calculating the measured statistics values for each model. Figure 4 also shows that HJ is decomposed into a build phase (Figure 4(b-1)) and a probe phase (Figure 4(b-2)) because each operation of HJ is executed sequentially and can be modeled separately in the cost calculation formula based on measurement results.

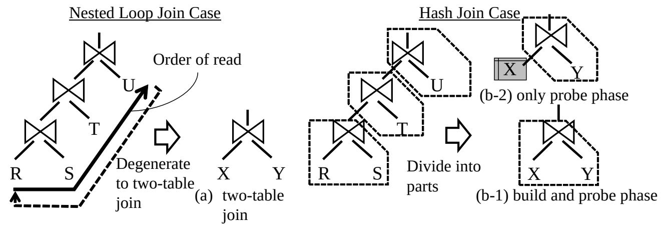


Figure 4. Degradation and split cost calculation method

## *C. Cost Calculation Formula*

Before considering the cost calculation formulas, we define the inputs and outputs as in Table II. The information input to the cost calculation formulas is recorded in the database for management as statistical information, which is collected generally by the DBMS when storing or updating the record. Information regarding memory latency and I/O response time is also required. This information can be measured with a simple benchmark program [10].

TABLE II. PARAMETER LIST FOR COST CALCULATION

| | |
|---|---|
| Input | Selectivity of outer table to join and number of records of tables |
| Output | Calculated cost expressed by number of CPU cycles |
| Parameters of cost calculation formulas | *Static information*: Memory latency and I/O response time<br>*Information obtained from measurement*: Relational formula between the input information and number of CPU cycles of the events on the right-hand side of (14) (e.g., slope and intercept if the input information and the number of cycles of the interested event can be linearly approximated.) |

In this section, we derive the cost calculation formula for NLJ and HJ in two tables (14) where each element of (14) is obtained as a function of the selectivity from the outer table and the number of records. The cost formula of NLJ

$$C_{NLJ\_Total}(P, R_O, R_I) = C_{NLJ\_ICacheMiss}(P, R_O, R_I) + C_{NLJ\_MP}(P, R_O, R_I) + C_{NLJ\_DCacheAcc}(P, R_O, R_I) \quad (15)$$

is obtained by combining (10)(11)(13)(14). The cost related to each element of the instruction cache miss, the branch misprediction, and the data reference are expressed by

$$\begin{aligned}
&C_{NLJ\_ICacheMiss}(P,R_O,R_I)\\
&\quad= M_{L2I}(P,R_O,R_I)\times L_{L2}\times BF_{L2I}\\
&\quad+ M_{LLCI}(P,R_O,R_I)\times L_{LLC}\times BF_{LLCI}\\
&\quad+ M_{memI}(P,R_O,R_I)\times L_{mem}\times BF_{memI} \quad (16)\\
&C_{NLJ\_MP}(P,R_O,R_I)\\
&\quad= M_{MP}(P,R_O,R_I)\times L_{MP}\times BF_{MP}(P,R_O,R_I) \quad (17)\\
&C_{NLJ\_DCacheAcc}(P,R_O,R_I)\\
&\quad= M_{L1D}(P,R_O,R_I)\times L_{L1}\times BF_{L2D}(P,R_O,R_I)\\
&\quad+ M_{L2D}(P,R_O,R_I)\times L_{L2}\times BF_{L2D}(P,R_O,R_I)\\
&\quad+ M_{LLCD}(P,R_O,R_I)\times L_{LLC}\times BF_{LLCD}(P,R_O,R_I)\\
&\quad+ M_{memD}(P,R_O,R_I)\times L_{mem}\times BF_{mem}(P,R_O,R_I) \quad (18)
\end{aligned}$$

respectively. The structure of the cost calculation formulas is basically a product-sum formula of the number of occurrences of the event, its latency, and the correction coefficient. The number of data references from the L1D cache, the L2 cache, the LLC cache, and the main memory ($M_{L1D}$, $M_{L2}$, $M_{LLC}$, and $M_{mem}$), the number of branch mispredictions ($M_{MP}$), and the blocking factor $BF$ are expressed as a function of the selectivity $P$ and the number of rows of the table, $R_O$, $R_I$. The data references include L1D hits because they target all data accesses.

The cost calculation formula of HJ is obtained in the same way as NLJ:

$$\begin{aligned}
&C_{Phase\_Total}(P,R) = C_{Phase\_ICacheMiss}(P,R)\\
&\quad+ C_{Phase\_MP}(P,R) + C_{Phase\_DCacheAcc}(P,R) \quad (19)\\
&C_{Phase\_ICacheMiss}(P,R)\\
&\quad= M_{L2I}(P,R)\times L_{L2}\times BF_{L2I}(P,R)\\
&\quad+ M_{LLCI}(P,R)\times L_{LLC}\times BF_{LLCI}(P,R)\\
&\quad+ M_{memI}(P,R)\times L_{mem}\times BF_{memI}(P,R) \quad (20)\\
&C_{Phase\_MP}(P,R)\\
&\quad= M_{MP}(P,R)\times L_{MP}\times BF_{MP}(P,R) \quad (21)\\
&C_{Phase\_DCacheAcc}(P,R)\\
&\quad= M_{L1D}(P,R)\times L_{L1}\times BF_{L2D}(P,R)\\
&\quad+ M_{L2D}(P,R)\times L_{L2}\times BF_{L2D}(P,R)\\
&\quad+ M_{LLCD}(P,R)\times L_{LLC}\times BF_{LLCD}(P,R)\\
&\quad+ M_{memD}(P,R)\times L_{mem}\times BF_{memD}(P,R) \quad (22)
\end{aligned}$$

where

$$\{Phase, R\} = \begin{cases} \{Build, R_O\} & \text{build phase} \\ \{Probe, R_I\} & \text{probe phase} \end{cases}$$

In the build phase, cache and main memory references, branch misprediction, and the blocking factor are expressed as functions of the selectivity $P$ and the number of records of the outer table ($R_O$). In the probe phase, they are expressed as functions of the selectivity $P$ and the number of records of the inner table ($R_I$).

The aim of this paper is to improve the accuracy of the CPU cost calculation. Therefore, we use a method to statistically obtain the parameters of the calculation formula from the measured values using the performance monitor. One of the parameters, the memory latency, depends on the hardware configuration, which includes the number of CPUs, the slot position in which the main memory modules are installed, etc. According to the literature [11], the memory access concentration is low when executing analytic queries such as the TPC-H benchmark and does not increase the memory latency.

## III. Evaluation of Obtaining Parameters of the Cost Formula

To obtain the parameters in Table II, actual measurements are made. The measurement environment is shown in Table III. We used Westmere CPUs as they are the same architecture as Nehalem. The servers are equipped with two CPUs. The main memory is connected to each CPU. The memory connected to one CPU is called local memory and the other is called remote memory. In general, such a memory architecture is known as non-uniform memory access (NUMA). The latency of local and remote memory is different. In this study, main memory modules are installed to only one CPU to simplify the examination of measurement results. An NVM Flash SSD is used as a disk device to store the database to improve the experimental efficiency. We used the open-source MariaDB [12] as the DBMS in this study as it supports multithreading and asynchronous I/O, can utilize the latest hardware characteristics, and, moreover, supports multiple join methods. In a precise sense, the NLJ that MariaDB supports is BNL (block nested loop join), which is an improvement on NLJ; however, in the condition of the query and index used in this study, it behaves like the general NLJ. The version of MariaDB used in this study does not select the effective join method automatically; it is specified based on the configuration parameters.

TABLE III. EVALUATION ENVIRONMENT

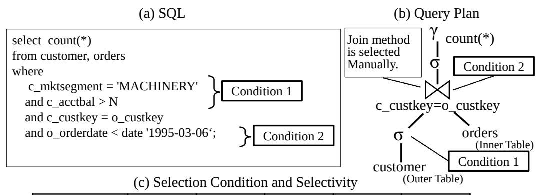
| | |
|---|---|
| CPU | Xeon L5630 2.13 GHz 4-core, LLC 12 MB [Westmere-EP]) $\times 2$ |
| Memory | DDR3 12 GB (4 GB $\times 3$) physically attached to only one CPU |
| Disk (DB) | PCIe NVMe Flash SSD 800 GB $\times 1$ (Note: max throughput suppressed by server's PCIe I/F(ver.1.0a), about 1/4 of max throughput) |
| Disk (OS) | SAS 10,000 rpm 600 GB, RAID5 (4 Data + 1 Parity) |
| OS | CentOS 6.6 (x64) |
| DBMS | MariaDB 10.1.8 with InnoDB storage engine |

The query to be evaluated and its measurement conditions are shown in Figure 5. In the SQL statement, we modified Query 3 of TPC-H for an evaluation of two-table join and extracted only join processing (Figure 5(a)). The size of database is scale factor (SF) 5 defined in the TPC-H specification. SF5 means that the total size of the database is 5 GB. To allow us to apply the proposed technology to the actual system, we used small-scale data to reduce the measurement time as much as possible. The indices of the database are created on the primary keys and the foreign keys are defined in the specification of TPC-H [13].

We changed the search condition of the query against the c_acctbal column of the outer table to change the selectivity of the data to be referenced (Figure 5(c)). As for NLJ, the selectivity and the number of records of the inner table are changed (Figure 5(c) and (d)). The purpose of changing the selectivity is to change the total number of records accessed by the DBMS. In addition, the purpose of changing the number of records of the inner table is to change the number of records that have the same key as the record selected from the outer table. This means changing the length of the linked lists that have the key for join with the inner table. As for HJ, only the outer table is accessed in the build phase, and the number of records of the outer table is changed (Figure 5(e)). In the probe phase, only the inner table is accessed, and the number of records of the inner table is changed (Figure 5(d)).

The CPU performance counter data is collected by using Intel® Vtune™ Amplifier XE. We refer to the literature [7] for a description of the content of those counters. The measured data is mainly related to the number of accesses to cache and main memory, the state of the pipeline such as the number of stall cycles, and the number of cache hits or misses.

It is necessary to analyze not only CPU time but also I/O operation time to estimate the whole execution time of a query (1). We measure the I/O count and response time using systemtap and construct I/O cost calculation formulas by analyzing the relation between I/O and the selectivity or the number of records.



```
select  count(*)
from customer, orders
where
   c_mktsegment = 'MACHINERY'
  and c_acctbal > N
  and c_custkey = o_custkey
  and o_orderdate < date '1995-03-06‘;
```

(c) Selection Condition and Selectivity

| *N* | 9998 | 9978 | 9798 | 9200 | 9000 |
|---|---|---|---|---|---|
| Selectivity (Condition 1) | $3.62\times10^{-05}$ | $4.00\times10^{-04}$ | $3.67\times10^{-03}$ | $1.45\times10^{-02}$ | $1.82\times10^{-02}$ |

(d) Inner Table

| | | | | |
|---|---|---|---|---|
| Number of Records | 7,500,000 | 5,625,000 | 3,750,000 | 1,875,000 |

(e) Outer Table

| | | | | |
|---|---|---|---|---|
| Number of Records | 750,000 | 562,500 | 375,000 | 187,500 |

Figure 5. Target query of measurement and cost estimation

## IV. Measurement Results and Cost Calculation Formulas

In this paper, we investigate the relations among the selectivity, the number of instructions, the number of events related to memory reference, and the number of branch mispredictions. Regarding NLJ, it is expected that the number of instructions and the number of memory references will increase because the number of records accessed by the DBMS increases in proportion to the increase in the selectivity. Based on the assumptions, we now analyze the measurement results and create formulas using linear regression. Regarding HJ, all of the records of the outer table or inner table are accessed in both the build phase and probe phase. The cost formulas are presumed to not have selectivity as a variable. We analyze the measurement results based on this presumption.

The CPU cost calculation formulas are obtained through the following steps. First, the number of instructions, references of each cache memory and main memory, and branch mispredictions are analyzed using regression analysis, and the regression models are created. In addition, the relation between the sum of the product of the references to each memory and its latency and $C_{ICacheMiss}$ (10) and $C_{DCacheAcc}$ (13) are modeled. Here $C_{MP}$ (11) is obtained from the product of the number of pipeline stages of the front-end, which is 12 in Nehalem, and the number of mispredictions from the measurement results. Each value of memory latency is referred to [14]. The number of disk I/O is modeled using the measured I/O access count and I/O response time. Finally, the cost calculation formulas are evaluated from the point of the accuracy of intersection of two join methods ($X_{cross}$ in Figure 2) with the conventional method.

Figure 6(1) shows the relation between the number of records the DBMS accessed and load instructions. Figure 6(7) shows the relation between the total number of accessed records and the number of instructions. The number of records is the product of the number of outer table records, the number of inner table records, and selectivity. The dotted line is the linear regression line, and its slope and intercept are shown in Table IV. The coefficient of determination ($R^2$) is near 1 and the $P$ value on the $F$ test is less than 0.05. Therefore, the regression model is highly accurate. The slope and the intercept are used for creating the cost calculation model. Figure 6(2) and (8) show the relation between the number of instructions executed by the DBMS and the number of L1 cache hits. Figure 6(3)–(6) and (9)–(12) show the relation between the number of accesses to L2, LLC, and main memory accesses and the number of cache misses of the upper-level cache. These relations can be linearly approximated because each $R^2$ is near 1 and each $P$ value is less than 0.05 in Table IV. In this paper, a two-CPU server is used and the LLC and main memory are connected to each CPU. The LLC and main memory on the CPU on which BMS threads are running are called the *local LLC* and *local main memory*. The others are called *remote LLC* and *remote main memory*. The upper-level cache is the local LLC. There exist no references to the remote main memory because the main memory is connected to only one CPU. Figure 6(13) shows the relation between the number of records accessed for the join operation and the branch miss prediction cycles, $C_{MP}$. Figure 6(14) shows the relation between the product of the number of instruction accesses and latency, and the L1I miss cycles, $C_{ICacheMiss}$. Figure 6(15) shows the relation between the product of the number of data accesses and the latency, and the data cache and main memory access, $C_{DCacheAcc}$. Each graph can be approximated by a regression line because each $R^2$ is near 1 and each $P$ value is less than 0.05 in Table IV. Figure 7(13) shows the tendency of instructions, cache or main memory accesses, branch misprediction cycles, instruction cache miss cycles, and data cache access cycles. Each graph is linearly approximated by a regression line because each $R^2$ is near 1 and each $P$ value is less than 0.05 in Table IV. In particular, the slope of the regression line in Figure 6(2)–(5) and (9)–(11) and Figure 7(a2)–(a5), (a9)–(a11), (b2)–(b5), and (b9)–(b11) represents the cache hit rate because the definition of cache hit rate is the quotient of the number of cache hits and the number of cache references, and the upper-level cache miss becomes the lower-level cache reference.

Based on the above considerations, the formula for calculating the cost of join methods is

$$I = A1 \times R + B1 \tag{23}$$

$$M_{L1I} = A2 \times I + B2 \tag{24}$$

$$M_{L2I} = A3 \times (I - M_{L1I}) + B3$$

$$M_{LLLCI} = A4 \times (I - M_{L1I} - M_{L2I}) + B4 \tag{25}$$

$$M_{RLLCI} = A5 \times (I - M_{L1I} - M_{L2I} - M_{LLLCI}) + B5 \tag{26}$$

$$M_{LMMI} = A6 \times (I - M_{L1I} - M_{L2I} - M_{LLLCI}) + B6 \tag{27}$$

$$I_{Load} = A7 \times R + B7 \tag{28}$$

$$M_{L1D} = A8 \times I_{Load} + B8 \tag{29}$$

$$M_{L2D} = A9 \times (I - M_{L1DI}) + B9 \tag{30}$$

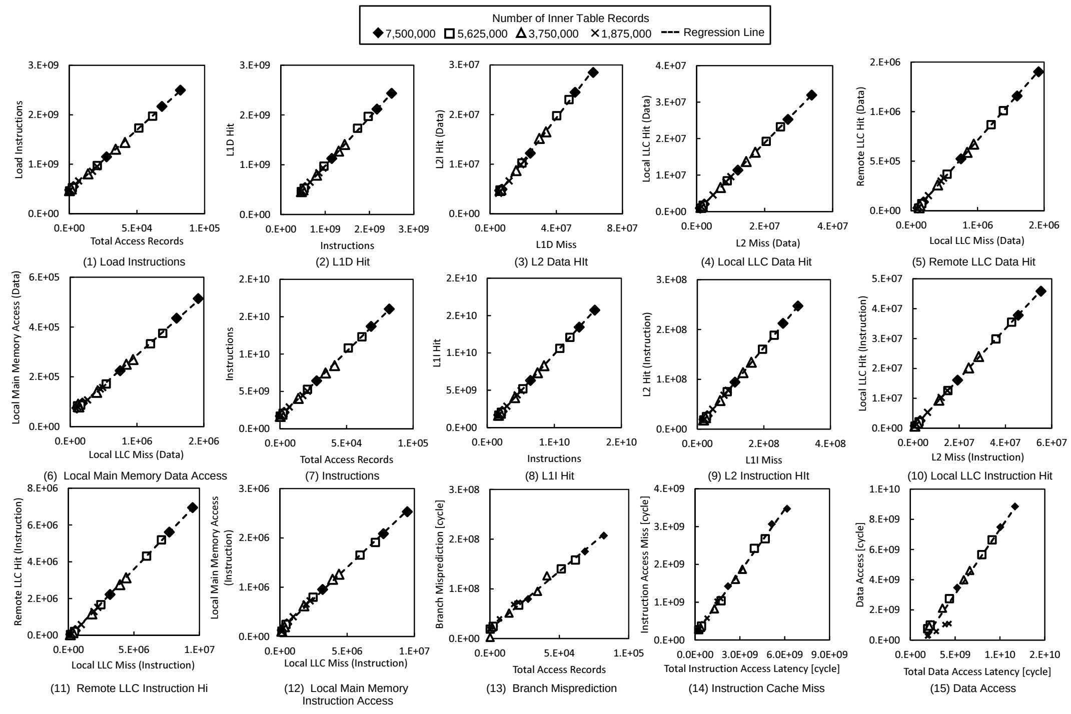


Figure 6. CPU event count on executing NLJ

$$M_{LLLCD} = A10 \times (I - M_{L1D} - M_{L2DI}) + B10 \tag{31}$$

$$M_{RLLCD} = A11 \times (I - M_{L1D} - M_{L2D} - M_{LLLCD}) + B11 \tag{32}$$

$$M_{LMMD} = A12 \times (I - M_{L1D} - M_{L2D} - M_{LLLCD}) + B12 \tag{33}$$

$$\begin{aligned} C_{ICacheMiss} = A13 \times (&M_{L2I} \times L_{L2} + M_{LLLCI} \times L_{LLLC} \\ &+ M_{RLLCI} \times L_{RLLC} + M_{LMMI} \times L_{LMM}) + B13 \end{aligned} \tag{34}$$

$$\begin{aligned} C_{DCacheAcc} = A14 \times (&M_{L1D} \times L_{L1} + M_{L2D} \times L_{L2} \\ &+ M_{LLLCD} \times L_{LLLC} + M_{RLLCD} \times L_{RLLC} \\ &+ M_{LMMD} \times L_{LMM}) + B14 \end{aligned} \tag{35}$$

$$C_{MP} = A15 \times R + B15 \tag{36}$$

where

$$R = \begin{cases} R_O \times R_I \times P & \text{NLJ} \\ R_O & \text{Build phase of HJ} \\ R_I & \text{Probe phase of HJ} \end{cases}$$

Table IV lists the definitions of the parameters given in (23)-(36). In the case of NLJ, the calculation formula of the number of the disk I/Os is created by using the regression line shown in Figure 8(a). The measured I/O response time ($io_responcetime$) was 154 $\mu$s. The I/O cost of NLJ is

$$io\_cost = A16 \times R_O \times R_I \times P \times io\_responce\_time + B16 \tag{37}$$

However, in the case of HJ, the ratio of the processing time of disk I/O and the query execution time of HJ is less than 1% in Figure 8(b). In this paper, the cost calculation formula is composed of only the CPU cost and the disk I/O cost.

To evaluate the cost calculation formulas, we used a larger TPC-H database than the database used for measurement, SF100, and chose a combination of the following two tables, *customer* and *orders*, *supplier* and *lineitem*, and *part*

TABLE IV. SLOPE AND INTERCEPT OF THE REGRESSION MODELS

| Type | | Slope | | Intercept | R$^2$ | $P$ value on $F$ test | Reference |
|---|---|---|---|---|---|---|---|
| NLJ | A1 | $1.745 \times 10^{5}$ | B1 | $1.64 \times 10^{9}$ | $9.99 \times 10^{-1}$ | $1.30 \times 10^{-29}$ | Figure 6(1) |
| | A2 | $9.802 \times 10^{-1}$ | B2 | $1.26 \times 10^{7}$ | $1.00$ | $2.18 \times 10^{-58}$ | Figure 6(2) |
| | A3 | $8.077 \times 10^{-1}$ | B3 | $2.68 \times 10^{6}$ | $1.00$ | $2.91 \times 10^{-40}$ | Figure 6(3) |
| | A4 | $8.318 \times 10^{-1}$ | B4 | $5.23 \times 10^{4}$ | $1.00$ | $3.93 \times 10^{-39}$ | Figure 6(4) |
| | A5 | $7.425 \times 10^{-1}$ | B5 | $-1.18 \times 10^{5}$ | $1.00$ | $3.77 \times 10^{-34}$ | Figure 6(5) |
| | A6 | $2.575 \times 10^{-1}$ | B6 | $1.18 \times 10^{5}$ | $9.98 \times 10^{-1}$ | $7.05 \times 10^{-26}$ | Figure 6(6) |
| | A7 | $2.464 \times 10^{4}$ | B7 | $4.63 \times 10^{8}$ | $9.99 \times 10^{-1}$ | $5.97 \times 10^{-31}$ | Figure 6(7) |
| | A8 | $9.723 \times 10^{-1}$ | B8 | $7.05 \times 10^{6}$ | $1.00$ | $3.85 \times 10^{-53}$ | Figure 6(8) |
| | A9 | $4.342 \times 10^{-1}$ | B9 | $1.95 \times 10^{6}$ | $9.99 \times 10^{-1}$ | $5.17 \times 10^{-28}$ | Figure 6(9) |
| | A10 | $9.442 \times 10^{-1}$ | B10 | $-2.87 \times 10^{4}$ | $1.00$ | $2.06 \times 10^{-44}$ | Figure 610) |
| | A11 | $7.609 \times 10^{-1}$ | B11 | $-4.84 \times 10^{4}$ | $1.00$ | $2.80 \times 10^{-35}$ | Figure 6(11) |
| | A12 | $2.391 \times 10^{-1}$ | B12 | $4.84 \times 10^{4}$ | $9.98 \times 10^{-1}$ | $3.10 \times 10^{-26}$ | Figure 6(12) |
| | A13 | $5.526 \times 10^{-1}$ | B13 | $1.44 \times 10^{8}$ | $9.98 \times 10^{-1}$ | $9.85 \times 10^{-26}$ | Figure 6(13) |
| | A14 | $8.595 \times 10^{-1}$ | B14 | $-1.25 \times 10^{9}$ | $9.67 \times 10^{-1}$ | $9.00 \times 10^{-15}$ | Figure 6(14) |
| | A15 | $2.321 \times 10^{3}$ | B15 | $1.92 \times 10^{7}$ | $9.90 \times 10^{-1}$ | $1.35 \times 10^{-19}$ | Figure 6(15) |
| HJ | A1 | $2.045 \times 10^{3}$ | B1 | $1.58 \times 10^{7}$ | $1.00$ | $4.21 \times 10^{-40}$ | Figure 7(a1) |
| Build | A2 | $9.879 \times 10^{-1}$ | B2 | $2.53 \times 10^{5}$ | $1.00$ | $2.19 \times 10^{-61}$ | Figure 7(a2) |
| | A3 | $9.708 \times 10^{-1}$ | B3 | $-7.48 \times 10^{4}$ | $1.00$ | $1.79 \times 10^{-49}$ | Figure 7(a3) |
| | A4 | $9.191 \times 10^{-1}$ | B4 | $-6.57 \times 10^{4}$ | $9.99 \times 10^{-1}$ | $7.76 \times 10^{-31}$ | Figure 7(a4) |
| | A5 | $3.317 \times 10^{-1}$ | B5 | $-1.64 \times 10^{4}$ | $9.29 \times 10^{-1}$ | $8.38 \times 10^{-12}$ | Figure 7(a5) |
| | A6 | $6.683 \times 10^{-1}$ | B6 | $1.64 \times 10^{4}$ | $9.82 \times 10^{-1}$ | $4.49 \times 10^{-17}$ | Figure 7(a6) |
| | A7 | $6.099 \times 10^{2}$ | B7 | $2.85 \times 10^{5}$ | $9.99 \times 10^{-1}$ | $4.46 \times 10^{-30}$ | Figure 7(a7) |
| | A8 | $9.902 \times 10^{-1}$ | B8 | $-1.89 \times 10^{3}$ | $1.00$ | $1.05 \times 10^{-57}$ | Figure 7(a8) |
| | A9 | $8.033 \times 10^{-1}$ | B9 | $-2.39 \times 10^{4}$ | $1.00$ | $6.00 \times 10^{-33}$ | Figure 7(a9) |
| | A10 | $9.042 \times 10^{-1}$ | B10 | $1.58 \times 10^{2}$ | $1.00$ | $8.94 \times 10^{-46}$ | Figure 7(a10) |
| | A11 | $2.131 \times 10^{-1}$ | B11 | $-2.74 \times 10^{3}$ | $9.80 \times 10^{-1}$ | $1.13 \times 10^{-16}$ | Figure 7(a11) |
| | A12 | $7.869 \times 10^{-1}$ | B12 | $2.74 \times 10^{3}$ | $9.98 \times 10^{-1}$ | $8.16 \times 10^{-27}$ | Figure 7(a12) |
| | A13 | $1.226$ | B13 | $-8.36 \times 10^{6}$ | $9.98 \times 10^{-1}$ | $1.55 \times 10^{-25}$ | Figure 7(a13) |
| | A14 | $3.691 \times 10^{-1}$ | B14 | $2.02 \times 10^{7}$ | $1.00$ | $6.83 \times 10^{-32}$ | Figure 7(a14) |
| | A15 | $2.351 \times 10^{1}$ | B15 | $5.12 \times 10^{5}$ | $9.97 \times 10^{-1}$ | $2.86 \times 10^{-24}$ | Figure 7(a15) |
| HJ | A1 | $1.900 \times 10^{3}$ | B1 | $2.33 \times 10^{7}$ | $1.00$ | $3.46 \times 10^{-46}$ | Figure 7(b1) |
| Probe | A2 | $9.883 \times 10^{-1}$ | B2 | $3.88 \times 10^{5}$ | $1.00$ | $3.69 \times 10^{-62}$ | Figure 7(b2) |
| | A3 | $9.750 \times 10^{-1}$ | B3 | $-1.76 \times 10^{4}$ | $1.00$ | $6.81 \times 10^{-52}$ | Figure 7(b3) |
| | A4 | $8.131 \times 10^{-1}$ | B4 | $-2.44 \times 10^{4}$ | $1.00$ | $1.12 \times 10^{-41}$ | Figure 7(b4) |
| | A5 | $9.455 \times 10^{-1}$ | B5 | $-2.44 \times 10^{4}$ | $1.00$ | $8.30 \times 10^{-36}$ | Figure 7(b5) |
| | A6 | $5.449 \times 10^{-2}$ | B6 | $2.44 \times 10^{4}$ | $9.56 \times 10^{-1}$ | $1.15 \times 10^{-13}$ | Figure 7(b6) |
| | A7 | $5.763 \times 10^{2}$ | B7 | $-3.57 \times 10^{7}$ | $9.99 \times 10^{-1}$ | $6.14 \times 10^{-27}$ | Figure 7(b7) |
| | A8 | $9.892 \times 10^{-1}$ | B8 | $-3.89 \times 10^{5}$ | $1.00$ | $6.68 \times 10^{-54}$ | Figure 7 (b8) |
| | A9 | $7.288 \times 10^{-1}$ | B9 | $1.58 \times 10^{5}$ | $9.88 \times 10^{-1}$ | $7.30 \times 10^{-19}$ | Figure 7(b9) |
| | A10 | $7.946 \times 10^{-1}$ | B10 | $2.81 \times 10^{4}$ | $1.00$ | $5.98 \times 10^{-33}$ | Figure 7 (b10) |
| | A11 | $9.341 \times 10^{-1}$ | B11 | $-6.04 \times 10^{4}$ | $1.00$ | $7.79 \times 10^{-34}$ | Figure 7 (b11) |
| | A12 | $6.595 \times 10^{-2}$ | B12 | $6.04 \times 10^{4}$ | $9.52 \times 10^{-1}$ | $2.69 \times 10^{-13}$ | Figure 7(b12) |
| | A13 | $1.503$ | B13 | $2.58 \times 10^{07}$ | $9.89 \times 10^{-1}$ | $5.80 \times 10^{-19}$ | Figure 7(b13) |
| | A14 | $3.576 \times 10^{-1}$ | B14 | $6.61 \times 10^{7}$ | $9.98 \times 10^{-1}$ | $1.75 \times 10^{-25}$ | Figure 7(b14) |
| | A15 | $2.761 \times 10^{1}$ | B15 | $-1.12 \times 10^{7}$ | $9.35 \times 10^{-1}$ | $3.83 \times 10^{-12}$ | Figure 7(b15) |
| NLJ (I/O) | A16 | $1.016$ | B16 | $2.52 \times 10^{3}$ | $1.00$ | $1.85 \times 10^{-15}$ | Figure 8(a) |
| HJ (I/O) | A16 | $0.000$ | B16 | $0.000$ | N/A | N/A | N/A |

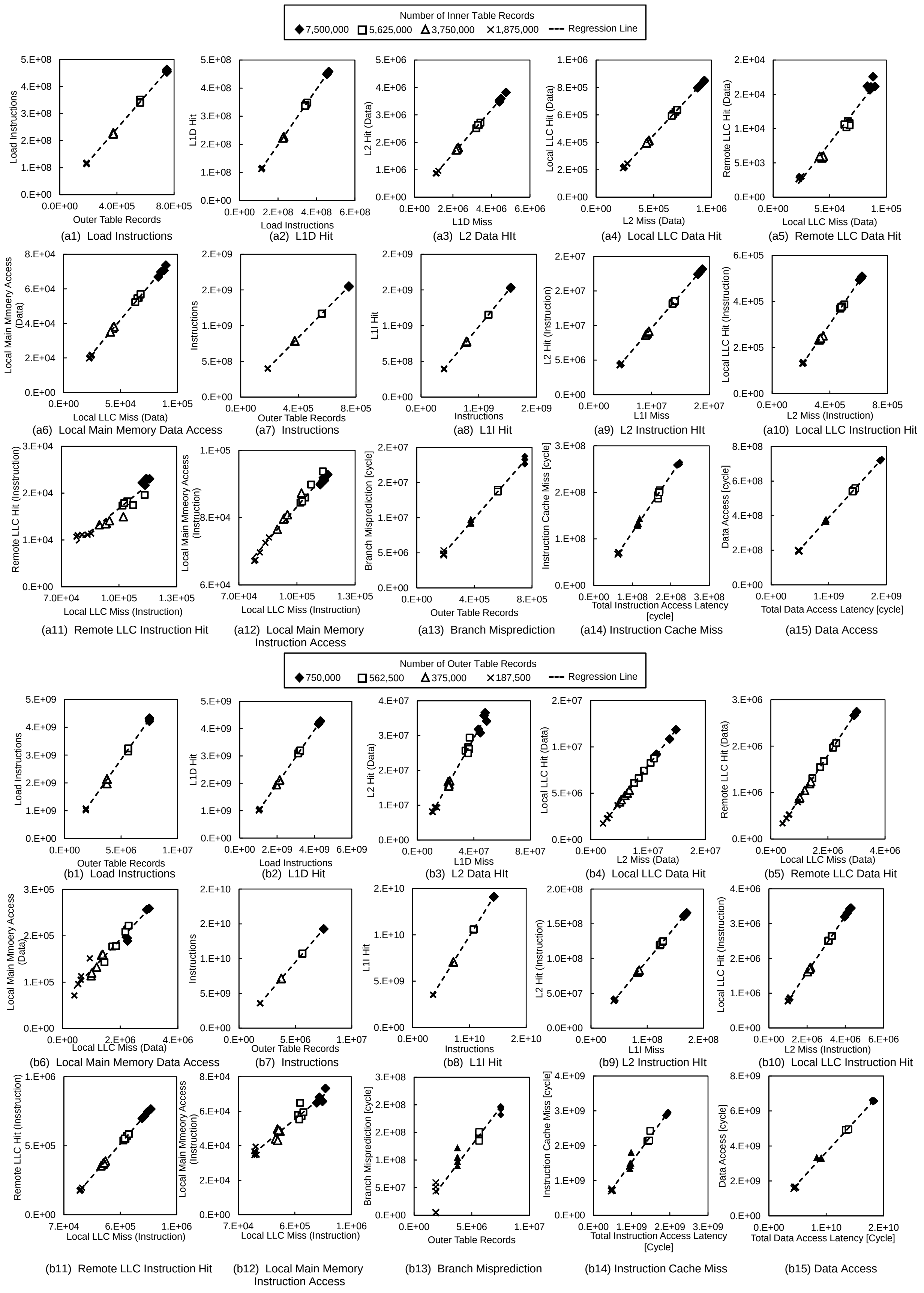


Figure 7. CPU event count on executing HJ

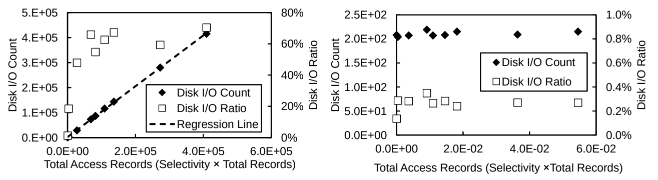


Figure 8. Number of disk I/O and disk I/O processing time per query execution time

and *lineitem*. The parameters setting of the cost calculation formulas is generated from the measurement values when joining *customer* and *orders*, whose size is SF5. The I/O processing time is added to allow a comparison with the query execution time. The proposed cost calculation method is compared with the measured query execution time and the conventional method (2)(3). We evaluate whether the selectivity where the join method is switched can be estimated accurately. However, because the conventional method does not support HJ, single table scans of the outer table and inner table are used. Moreover, MariaDB, as used in this experiment, cannot use the function to automatically select the join method, and only the join method set by the user is selected. The goal of this study is to accurately find the intersection point of the NLJ and HJ graphs. As a result, in all of the cases evaluated in this study, the proposed method was able to find the intersection point with an accuracy of one significant figure or better compared with the conventional method (Figure 9).

## V. DISCUSSION

In the acquisition of measurement data for constructing the cost calculation formula, since the type of counters that the hardware monitor can collect at one time is limited to four, it is necessary to measure many times in order to perform an accurate measurement of 40 events. Therefore, a certain amount of time must be secured for measurement. For example, it takes about 5 hours and 30 minutes for the measurement of this study. From the point of securing time for measurement and the point that the CPU cost calculation formula does not need to change the CPU cost calculation formula unless there is a change in the hardware configuration or join operation codes of DBMS, it is appropriate to create the proposed CPU cost calculation formula at integrating or updating a system. Next, in the use of the cost calculation formula, the proposed CPU cost formula is used in the optimization process to be executed before executing a query. The CPU cost of executing the query is calculated from the number of records to be searched. As shown in the reference [15] [16], in a general DBMS, histograms representing the relationship between the attribute value and the appearance frequency are automatically acquired when inserting or updating records. From the histogram and the condition of the where clause of the query, it is possible to estimate the number of records accessed by the DBMS. In this way, CPU costs can be calculated with only the data already acquired by the DBMS, so cost can be calculated by the cost calculation formula before query execution.

In this study, we have proposed a cost calculation method for the in-memory DBMS using a disk-based DBMS. The calculation formulas have been created using the data measured by the CPU-embedded performance monitor. The study revealed that the proposed method estimated the intersection point of the join methods more accurately than the conventional method. We used TPC-H for measuring CPU activities. TPC-H has the advantage that it is easy to analyze the evaluation results because the distribution of data is uniform. However, the actual data has a skew in the distribution of keys. The premise of the technique in this paper is the accuracy of selectivity. Even if the distribution of data varies, if the selectivity is the same, the same measurement result is obtained. Since a general DBMS acquires attribute values and their distribution in a database in the form of a histogram when loading data to the database, the prerequisite for application of the proposed technique is considered to be satisfactory. However, it is necessary to develop a technique to derive histogram information and input it as an input parameter of the cost formulas.

Since this technique sets parameters based on actual measurements, it is difficult to deal with various patterns such as the presence or absence of indices and complicated queries. Although we have focused on the operation of all CPU cycles, it is necessary for practical use to simplify the model omitting some parameters. For the collection of statistical data, it is conceivable that actual measurement could be performed at the time of initial installation and parameter setting. However, when the code of the DBMS is modified, it is difficult to change in real time, so separate complementary technology is required. As a breakthrough measure, it is possible to reduce the amount of data to be verified and to reduce measurement points.

## VI. RELATED WORK

Evaluating CPU performance using the performance monitor for behavior analysis of a DBMS has long been performed. In particular, in the evaluation of the benchmark TPC-D for e decision support systems, the L1 miss and the processing delay due to L2 cache occupy a large part as the components of the CPI, and it is important in terms of performance. However, it is only used for bottleneck analysis [17].

There is research that applied a CPI calculation method focusing on a memory reference to cost calculation (5) for an in-memory database [18] [19]. This previous research targets DBMS that use the load/store type memory access (Figure 1(c)).

In this research, the number of cache hits or main memory accesses is predicted from the data access pattern of the database, and the cost is calculated as the product of the number of the cache hits or main memory accesses and the memory latency. Modeling of $CPI0$, which is the state that all data exists in the L1 cache, and modeling of instruction cache misses have not been considered in previous studies. Although not explicitly mentioned in the literature, it was presumed that it was impossible to reproduce and measure the state in which all instructions and data are on the L1 cache, which is the definition of $CPI0$, by means such as a CPU-embedded performance monitor.

## VII. CONCLUSIONS AND FUTURE WORK

In this study, we have proposed a cost calculation method for the in-memory DBMS using disk-based DBMS. We focused on a CPU pipeline architecture and classified CPU cycles into three types based on the characteristics of operation of the front-end and back-end. The calculation formulas are created using the data measured by the CPU-embedded performance monitor. In the evaluation, the difference in selectivity corresponding to the intersection points of NLJ and HJ between the calculated cost and the measured time was less than 1%;

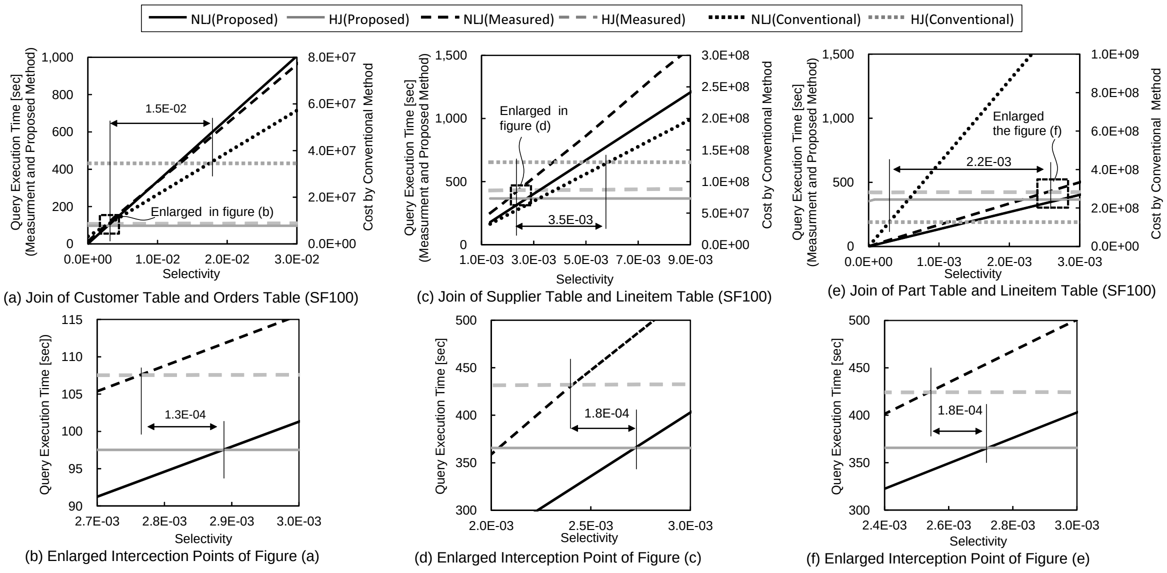


Figure 9. Comparison of measured results, the proposed cost model, and the conventional cost model

that is, the cost formulas can model the actual join operation with high accuracy. As a result, by applying the proposed cost calculation formulas, we can select the join method appropriately and reduce the risk of unexpected query execution delay to users of the DBMS. In the future, we will consider joins of three or more tables. Furthermore, we will evaluate different generation CPUs and analyze how the differences in CPU architecture affect the cost formulas and implement a DBMS that automatically distinguishes CPU differences from the analysis results and automatically corrects the parameters for cost calculation or the calculation model itself.

## REFERENCES

[1] A. Foong and F. Hady, “Storage as fast as rest of the system,” in 2016 IEEE 8th International Memory Workshop (IMW), May 2016, pp. 1–4.

[2] A. Rudoff, “Programming models to enable persistent memory,” Storage Developer Conference, SNIA, Santa Clara, CA, USA, September, 2012, https://www.snia.org/sites/default/orig/SDC2012/presentations/Solid_State/AndyRudoff_Program_Models.pdf [retrieved: March, 2017].

[3] A. Rudoff, “The impact of the nvm programming model,” Storage Developer Conference, SNIA, Santa Clara, CA, USA, September, 2013, https://www.snia.org/sites/default/files/files2/files2/SDC2013/presentations/GeneralSession/AndyRudoff_Impact_NVM.pdf [retrieved: March, 2017].

[4] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, “Access path selection in a relational database management system,” in Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD ’79. New York, NY, USA: ACM, 1979, pp. 23–34. [Online]. Available: http://doi.acm.org/10.1145/582095.582099

[5] W. Wu et al., “Predicting query execution time: Are optimizer cost models really unusable?” in Data Engineering (ICDE), 2013 IEEE 29th International Conference on, April 2013, pp. 1081–1092.

[6] O. Sandsta, “Mysql cost model,” http://www.slideshare.net/olavsa/mysql-optimizer-cost-model [retrieved: March, 2017], October 2014.

[7] D. Levinthal, “Performance analysis guide for intel core i7 processor and intel xeon 5500 processors,” Intel Performance Analysis Guide, vol. 30, 2009, p. 18.

[8] P. Apparao, R. Iyer, and D. Newell, “Towards modeling & analysis of consolidated cmp servers,” SIGARCH Comput. Archit. News, vol. 36, no. 2, May 2008, pp. 38–45. [Online]. Available: http://doi.acm.org/10.1145/1399972.1399980

[9] N. Hardavellas et al., “Database servers on chip multiprocessors: Limitations and opportunities,” in Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, USA, January 2007, pp. 79–87.

[10] L. McVoy et al., “lmbench: Portable tools for performance analysis.” in USENIX annual technical conference, San Diego, CA, USA, 1996, pp. 279–294.

[11] J. L. Lo et al., “An analysis of database workload performance on simultaneous multithreaded processors,” in ACM SIGARCH Computer Architecture News, vol. 26, no. 3. IEEE Computer Society, 1998, pp. 39–50.

[12] The mariadb foundation - ensuring continuity and open collaboration in the mariadb ecosystem. [Online]. Available: https://mariadb.org/ (2017)

[13] “TPC BENCHMARK$^{TM}$ H (decision support) standard specification revision 2.17.1, Transaction Processing Performance council (TPC),” http://www.tpc.org/tpc_documents_current_versions/pdf/tpch2.17.1.pdf [retrieved: March, 2017], 2014.

[14] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, “Memory performance and cache coherency effects on an intel nehalem multiprocessor system,” in 2009 18th International Conference on Parallel Architectures and Compilation Techniques, Sept 2009, pp. 261–270.

[15] A. Aboulnaga et al., “Automated statistics collection in db2 udb,” in Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30, ser. VLDB ’04. VLDB Endowment, 2004, pp. 1158–1169. [Online]. Available: http://dl.acm.org/citation.cfm?id=1316689.1316788

[16] I. Babae, “Engine-independent persistent statistics with histograms in mariadb,” Percona Live MySQL Conference and Expo 2013, April, 2013, https://www.percona.com/live/london-2013/sites/default/files/slides/uc2013-EIPS-final.pdf [retrieved: March, 2017].

[17] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, “Dbmss on a modern processor: Where does time go?” in VLDB” 99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK, no. DIAS-CONF-1999-001, 1999, pp. 266–277.

[18] S. Manegold, P. A. Boncz, and M. L. Kersten, “Optimizing database architecture for the new bottleneck: memory access,” The VLDB Journal, vol. 9, no. 3, 2000, pp. 231–246.

[19] S. Manegold, P. Boncz, and M. L. Kersten, “Generic database cost models for hierarchical memory systems,” in Proceedings of the 28th international conference on Very Large Data Bases. VLDB Endowment, 2002, pp. 191–202.