Measurement-based Cost Estimation Method for Multi-Table Join Operation in 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 the main memory to improve performance and increase capacity. In the near future, some in-memory database systems will use a non-volatile main memory as a durable medium instead of the 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 relative to the most suitable access path selected for a database query. Therefore, a high-precision cost calculation method for query execution is required. In particular, when the database system cannot select the proper join method, the query execution time increases. Accordingly, a database join operation cost model using statistical information measured by a performance monitor embedded in the CPU is proposed and the accuracy of estimating the change point of join methods is evaluated. The results show that the proposed method can improve the accuracy of cost calculations to more than 90% compared to the conventional method. In conclusion, the in-memory database system using the proposed cost calculation method can select the best join method.**

***Index Terms*—Non-volatile memory; In-memory database systems; Query optimization; Query execution cost.**

I. Introduction

This paper is the extended work of a paper presented at the MMEDIA 2017 conference [1]. Improving the performance and expanding the capacity of the non-volatile memory (NVM) is applicable to both high-speed disk drives and main memory units. Intel and Micron developed a NVM called 3D Xpoint memory [2] for such use. An NVM is implemented as a byte-addressable memory and assigned as part of the main memory space. An application programming interface (API) [3] [4] for accessing the NVM was proposed to make the development of applications easier. Roughly speaking, the API provides two types of access methods to the NVM from the software. The first is the "load/store type," which is the same method used to access the conventional main memory from user applications. The other is the "read/write type," which is the method used by existing input/output (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 a NVM to the 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 the NVM defined by the API for the NVM (Figure 1(b)). When accessing the database, the performance of the load/store type is better than the read/write type because the DBMS directly accesses the database without any I/O device emulation operation. Database administration operations (e.g., system configuration, backup, etc.) do not have to be changed, which means that it is easy for the administrators to introduce the in-memory database system.

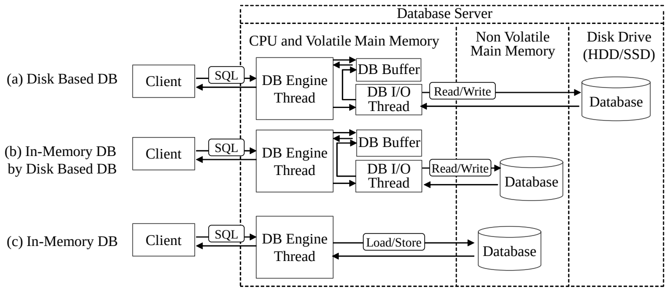


Fig. 1. Disk-based database and in-memory database

The DBMS encounters a problem when preparing for the execution of a query. In general, the DBMS performs several steps prior to query execution. First, it analyzes the query. Second, it creates multiple execution plans. Third, it estimates the query processing cost for each execution plan. Finally, it selects the minimum-cost execution plan from the 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 the join method that the DBMS selects. The DBMS estimates the cost of each join method using statistical information from the database, and chooses the method with the lowest cost. In general, the cost of a join operation is a function of the ratio of the extracted records to all the records. Hereafter, we refer to this ratio as the selectivity. In Figure 2, the selectivity is determined by condition x for column R.C in Figure 2(c). In Figure 2(c), two cost functions intersect at $\mathrm{X}_{\mathrm{cross}}$. Join method 2 must be chosen from the left side of $\mathrm{X}_{\mathrm{cross}}$, and join method 1 should be chosen from the right side of $\mathrm{X}_{\mathrm{cross}}$. If the DBMS cannot estimate the selectivity $\mathrm{X}_{\mathrm{cross}}$ accurately, it will choose the wrong join method.

On the other hand, the query execution cost ($cost$) is generally expressed as the sum of the central processing unit

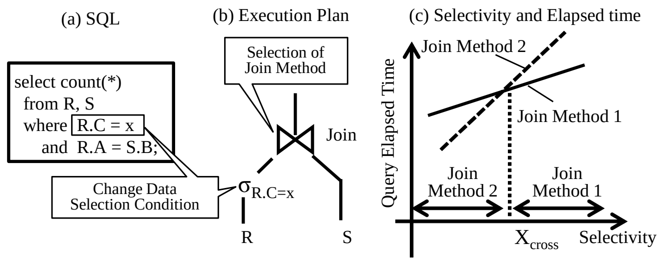


Fig. 2. Cost estimation problem for the selection of join methods

(CPU) cost (cpu_cost) and the I/O cost (io_cost) [5] [6]. The CPU cost is the CPU time, while 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 [7]. 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 selection conditions, such as a where-clause definition in SQL and $records\_per\_key(\mathrm{S})$, are 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 was 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. If the I/O cost itself ultimately disappears with a native in-memory database (Figure 1(c)), then it becomes necessary to accurately predict the CPU cost.

To improve the accuracy of CPU processing cost prediction, the estimation of CPU processing time must become more accurate than with 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 an instruction is completed. To estimate the latency with high accuracy, it is necessary to consider the hardware structure, such as instruction execution parallelism, cache miss ratio, and memory hierarchy. These problems cannot be solved by the software algorithm alone. In order to improve the accuracy of cost calculation, we focused on constructing a CPU operation model by considering the CPU architecture [1].

In general, two approaches exist for query cost calculation: white-box analysis [8] and black-box analysis [9]. In the white-box analytic approach, the cost calculation model for a single DBMS system is the sum of CPU cost and I/O cost. These cost calculation models are functions of the number of records accessed by DBMS. The black-box analysis approach does not compute the sum by using each operation cost like accessing records of tables, accessing I/O, etc., but calculates the cost using multiple regression, which analyzes the objective variable with the information that the user of the database ordinarily obtains (explanatory variable such as the cardinality of the table). In most of the open source and commercial DBMSs, the white-box analysis approach is used because of the ease of understanding the models. This study adopts the white-box analysis approach for the same reason. The black-box analysis approach can easily deal with any DBMS because it does not use DBMS-dependent information. However, its estimation accuracy worsens in cases where the value of cost is small [9]. Therefore, this study aims to calculate an accurate cost when the cost value is small, by modeling the CPU activities.

In this study, we propose a method based on statistical information on CPU operations to improve the accuracy of CPU cost estimation for in-memory databases applied to existing DBMSs (Figure 1(b)). Our method can be easily applied 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 the 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. Using this classification, we propose a CPU cycle modeling method that 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 join processing into four parts and estimate the join operation cost using a combination of these parts (Section II).
- Second, we analyze the trends or characteristics of the measured results for the join operation by using a performance monitor embedded in 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 conventional CPU cost estimation formula of MySQL (Section IV).

II. Proposed CPU Cost Model

In this section, we first analyze the CPU pipeline architecture and categorize pipeline events. Second, we propose the CPU operation cycle estimation method, which can express whole CPU process cycles by considering the categorized events. Third, 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 formulas for estimating each part of the join operation using statistical information measured by the performance monitor embedded in the CPU, and then combine these join part formulas to obtain 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 micro-operations (μOPs), increased reorder buffer entries, and increased instruction execution units, were added to the successor CPUs of Nehalem.

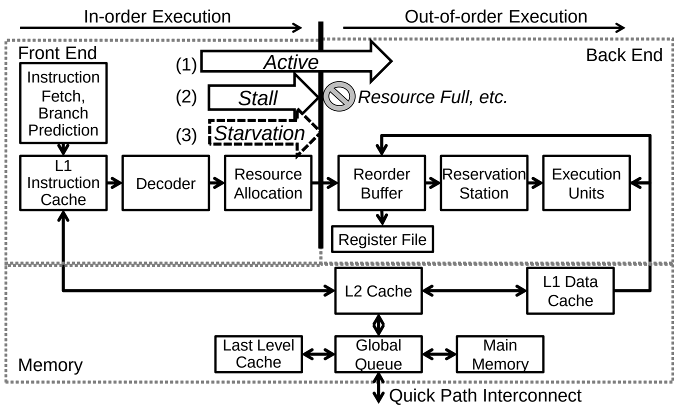


Fig. 3. Focus point of the CPU pipeline

The pipeline is composed of a front-end and back-end, as shown in Figure 3 [10]. The front-end fetches instructions from the L1 instruction cache (L1I) and decodes them into μOPs in-order. The term "in-order" means that a subsequent instruction cannot override the preceding instructions in the pipeline. After decoding the instructions, the front-end issues the μOPs to the back-end. Conversely, the back-end executes the μOPs in execution units that are out-of-order. The back-end can execute the μOPs in a different order than that issued by the front-end to improve the throughput of operating μOPs. An L1I miss causes the pipeline of the front-end to stall until the missing instruction is fetched from the lower level 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 μOPs. Such a condition is referred to as an *instruction-starvation state* (Figure 3(3)). There are cases in which the μOP issued in the front-end is not executed because of the saturation of the reorder buffer or reservation station in the back-end, or the data dependency of the preceding instructions. We refer to this state as a *stall state* (Figure 3(2)). In addition, we refer to the state in which the μOPs are issued without an *instruction-starvation state* or a *stall state* as an *active state*.

A summary of the notations related to CPU cost calculation to be used later in the study is presented in Table I before creating the CPU cost calculation model.

In this study, we focus on the boundary between the front-end and back-end in the CPU pipeline (Figure 3) to model the overall operation of the CPU. The μOPs are issued from front-end to back-end, and are stored in buffers, i.e., the reorder buffer and reservation station. The buffers allow us to change the processing order of μOPs from in-order to out-of-order across the boundary. The CPU-embedded performance monitor can measure events such as the saturation of buffers, de-queues from buffers by the completion of μOPs, and the existence of μOPs to issue to the back-end [10]. Any CPU cycle situation can be modeled by the performance monitor to analyze these events. Therefore, we propose a measurement-based estimation of the query execution cost. The *active state* is estimated from the number of events in which the μOP is issued without delay in the back-end buffer. The back-end buffer holds the μOPs until the execution of the μOPs is completed, and the μOPs are deleted from the buffer. The *stall state* is estimated from the number of events for which the buffer cannot receive μOPs. The *starvation state* is inferred from the event count where there are no μOPs to be issued to the back-end buffer. The total CPU cycle is composed of the *active state*, *stall state*, and *starvation state* cycles. Therefore, the following equation can be obtained:

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

The cycles per instruction (CPI) metric, which refers to the number of CPU clock cycles per instruction, is widely used for evaluating the CPU processing efficiency [11]. 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

$$\begin{aligned} CPI = CPI0 + \{ \sum_{i=2}^{LLC} (H_{Li} \times L_{Li} \times BF_{Li}) \\ + (H_{MM} \times L_{MM} \times BF_{MM}) \} \end{aligned} \tag{5}$$

where LLC denotes last level cache and means the lowest cache in the cache memory hierarchy; the blocking factor [11] 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, latency, and 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}):

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

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

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

TABLE I. NOTATIONS FOR THE CPU COST CALCULATION MODEL

Symbol		Description
I		Number of instructions to complete a query
I_{Load}		Number of load instructions
CPI		Cycle per instruction (CPI)
$CPI0$		Cycle per instruction (CPI) on the condition that all of instructions and data are stored in L1 cache
M_{events}		Number of events
	events	**Description**
	MM	References of instructions and data to main memory
	MMI	References of instructions to main memory
	MMD	References of data to main memory
	Li	References of instructions and data to Li cache
	LiI	References of instructions to Li cache
	MP	Branch mispredictions
	$LMMI$	References of instructions to local main memory
	$LMMD$	References of data to local main memory
	$LLLCI$	References of instructions to local LLC
	$LLLCD$	References of data to local LLC
	$RLLCI$	References of instructions to remote LLC
	$RLLCD$	References of data to remote LLC
L_{memory}		Latency of cache memory or main memory
	memory	**Description**
	MM	Main memory
	Li	Li Cache
	MP	Recovering latency from a branch misprediction
	LMM	Local main memory
	$LLLC$	Local LLC
	$RLLC$	Remote LLC
BF_{events}		Blocking factor of $events$
	events	**Description**
	MM	References of instructions and data to main memory
	MMI	References of instructions to main memory
	MMD	References of data to main memory
	Li	References of instructions and data to Li cache
	LiI	References of instructions to Li cache
	LiD	References of data to Li cache
	MP	Branch misprediction and instruction cache miss occur simultaneously
H_{memory}		Ratio of $memory$ references to instructions ($H_{memory} = M_{mememory}/I$)
	memory	**Description**
	MM	References to main memory
	Li	References to Li cache memory
	LiI	References of instructions to Li cache
	LiD	References of data to Li cache
C_{state}		CPU cycles in $state$ during executing a query
	state	**Description**
	$Total$	Total of all states
	$Active$	Not occurring stall
	$Stall$	Stall of CPU pipeline
	$Starvation$	Starvation of instructions to issue
	$ICacheMiss$	CPU cycles from occurrence of L1I miss until the acquisition of an instruction from other cache or the main memory
	$DCacheAcc$	CPU cycles in *active state*
	MP	Total CPU cycles when recovering from branch mispredictions
C_{join_state}		CPU cycles of $join$ in $state$
	join	**Description**
	NLJ	Nested Loop Join
	$Build$	Build phase of Hash Join
	$Probe$	Probe phase of Hash Join
	$CmdBld$	Combination build phase
	$CmdPrb$	Combination probe phase
	state	**Description**
	$ICacheMiss$	CPU cycles from occurrence of L1I miss until the acquisition of an instruction from other cache or the main memory
	$DCacheAcc$	CPU cycles in *active state*
	MP	Total CPU cycles when recovering from branch mispredictions
$RC_{I\_total(n)}$		Total number of accessed records in n inner tables and entries of indexes
P		Selectivity of the tables for join
P_O		Selectivity of the outer table
P_{Ik}		Selectivity of the inner table$k$
R_O		Number of records in outer table
R_{Ik}		Number of records in inner table k ($k = 1, 2, \cdots$)
$R_{I\_total(n)}$		Total number of accessed records in n inner tables
$RC_{I\_total(n)}$		Total number of accessed records in n inner tables and entries of indexes

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

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.

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

$$C_{ICacheMiss} = \sum_{Li=L2}^{LLC} (M_{LiI} \times L_{Li} \times BF_{LiI}) + (M_{MMI} \times L_{MM} \times BF_{MMI}) \tag{10}$$

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 the operations after instruction cache misses and data cache misses are the same. Only the terms relating to branch misprediction are defined.

$$C_{MP} = M_{MP} \times L_{MP} \times BF_{MP} \tag{11}$$

According to previous research [12], 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; thus the instructions are executed sequentially in query execution. In addition, because the indices and tables of the database are usually implemented with list or tree structures, the next reference address becomes clear only after the stored data that the pointer refers to is read out. In particular, the characteristics of such a memory reference in the list structure are applied to a benchmark program for measuring memory latency [13]. Therefore, the *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, the CPU cycles in the *active state* and *stall state* can be integrated as $C_{DCacheAcc}$

$$C_{DCacheAcc} = C_{Active} + C_{Stall} \tag{12}$$

$$C_{DCacheAcc} = \sum_{Li=L1}^{LLC} (M_{LiD} \times L_{Li} \times BF_{LiD}) + (M_{MMD} \times L_{MM} \times BF_{MMD}) \tag{13}$$

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

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

$$C_{Total} = C_{DCacheAcc} + C_{ICacheMiss} + C_{MP} \tag{14}$$

In this study, 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 operating 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 study focuses on 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 for multiple tables using the cost estimation function with two typical NLJs (Figure 4(a)), which is a function of the number of total records to be referenced in the multitable join. NLJ and HJ are regarded as part of our proposed cost estimation method. In this work, 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 then calculating the measured statistical 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.

When more than three tables are joined, the DBMS optimizer chooses a combination of different join methods for executing a query. Figure 5 shows a combination of HJ and NLJ. The first table to operate a join is called the "outer table," while the other tables are called "inner table." In addition, the inner tables are called "inner table1" and "inner table2" according to the joining order. A combination of different join methods is divided into an HJ build phase (Figure 5 (c-1)). The cost model of (c-1) is the same as (b-1). However, the cost model of (c-2) is different from the one mentioned above. It is presumed that the HJ probe phase and NLJ cannot be divided because the DBMS repeatedly searches one record in table Y using the hash table X, and searches table Z by NLJ. The (c-1) phase is called the "combination build phase" and the (c-2) phase is called the "combination probe phase."

C. Cost Calculation Formula

Before considering the cost calculation formulas, we define the inputs and outputs as listed in Table II. The information input into the cost calculation formulas is recorded in the database for management as statistical information, and is collected generally by the DBMS when storing or updating

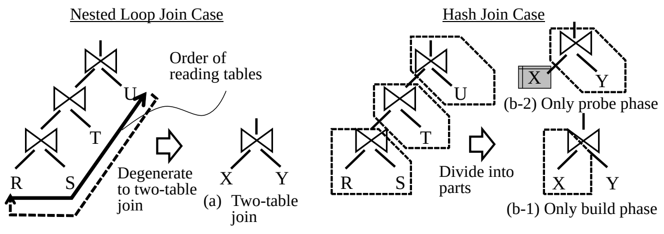


Fig. 4. Degradation and split cost calculation method

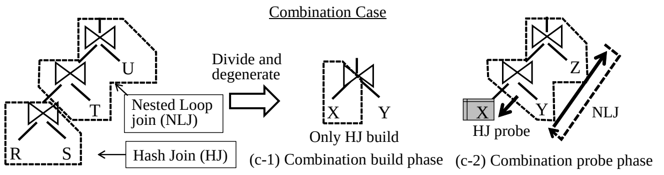


Fig. 5. NLJ after HJ Case

the record. Information regarding memory latency and I/O response time is also required. This information can be measured with a simple benchmark program [13].

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 *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 event of interest can be linearly approximated.)

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

$$\begin{aligned}C_{NLJ\_Total}(P, P_{Ik}, R_O, R_{Ik}) &= C_{NLJ\_ICacheMiss}(P_O, P_{Ik}, R_O, R_{Ik}) \\ &+ C_{NLJ\_MP}(P_O, P_{Ik}, R_O, R_{Ik}) \\ &+ C_{NLJ\_DCacheAcc}(P_O, P_{Ik}, R_O, R_{Ik}) \quad (15)\end{aligned}$$

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

$$\begin{aligned}C_{NLJ\_ICacheMiss}(P_O, P_{Ik}, R_O, R_{Ik}) &= M_{L2I}(P_O, P_{Ik}, R_O, R_{Ik}) \times L_{L2} \times BF_{L2I} \\ &+ M_{LLCI}(P_O, P_{Ik}, R_O, R_{Ik}) \times L_{LLC} \times BF_{LLCI} \\ &+ M_{MMI}(P_O, P_{Ik}, R_O, R_{Ik}) \times L_{MM} \times BF_{MMI} \quad (16)\end{aligned}$$

$$C_{NLJ\_MP}(P_O,P_{Ik},R_O,R_{Ik}) \\ = M_{MP}(P_O,P_{Ik},R_O,R_{Ik}) \times L_{MP} \times BF_{MP}(P_O,R_O,R_{Ik}) \quad (17)$$

$$C_{NLJ\_DCacheAcc}(P_O,P_{Ik},R_O,R_{Ik}) \\ = M_{L1D}(P_O,P_{Ik},R_O,R_{Ik}) \times L_{L1} \times BF_{L2D}(P_O,P_{Ik},R_O,R_{Ik}) \\ + M_{L2D}(P,R_O,R_{Ik}) \times L_{L2} \times BF_{L2D}(P_O,P_{Ik},R_O,R_{Ik}) \\ + M_{LLCD}(P,R_O,R_{Ik}) \times L_{LLC} \times BF_{LLCD}(P_O,P_{Ik},R_O,R_{Ik}) \\ + M_{MMD}(P,R_O,R_{Ik}) \times L_{MM} \times BF_{MM}(P_O,P_{Ik},R_O,R_{Ik}) \quad (18)$$

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, L2 cache, LLC cache, main memory (M_{L1D}, M_{L2}, M_{LLC}, and M_{MM}), number of branch mispredictions (M_{MP}), and blocking factor BF are expressed as a function of the selectivity P_O, P_{Ik}, and the number of rows of the table R_O, R_{Ik}. The cost of the instruction reference $C_{NLJ\_ICacheMiss}$ does not include L1I hits because it means the L1I cache miss penalty. However, the cost of the data reference $C_{NLJ\_DCacheAcc}$ includes L1D hits because the data reference includes all of the data access.

The cost calculation formula of HJ is obtained in the same way as that of NLJ with selectivity P

$$C_{Phase\_Total}(P,R) = C_{Phase\_ICacheMiss}(P,R) \\ + C_{Phase\_MP}(P,R) + C_{Phase\_DCacheAcc}(P,R) \quad (19)$$

$$C_{Phase\_ICacheMiss}(P,R) \\ = M_{L2I}(P,R) \times L_{L2} \times BF_{L2I}(P,R) \\ + M_{LLCI}(P,R) \times L_{LLC} \times BF_{LLCI}(P,R) \\ + M_{MMI}(P,R) \times L_{MM} \times BF_{MMI}(P,R) \quad (20)$$

$$C_{Phase\_MP}(P,R) \\ = M_{MP}(P,R) \times L_{MP} \times BF_{MP}(P,R) \quad (21)$$

$$C_{Phase\_DCacheAcc}(P,R) \\ = M_{L1D}(P,R) \times L_{L1} \times BF_{L2D}(P,R) \\ + M_{L2D}(P,R) \times L_{L2} \times BF_{L2D}(P,R) \\ + M_{LLCD}(P,R) \times L_{LLC} \times BF_{LLCD}(P,R) \\ + M_{MMD}(P,R) \times L_{MM} \times BF_{MMD}(P,R) \quad (22)$$

where

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

In the build phase, the cache and main memory references, branch misprediction, and blocking factor are expressed as functions of selectivity P and the number of records of the outer table (R_O). In the probe phase, these are expressed as functions of selectivity P and the number of records of the inner table (R_{Ik}). For a combination case like Figure 5(c-1), the cost formula of the combination build phase can be created with reference to the cost formula of the HJ build phase.

$$C_{CmbBld\_Total}(P_O,R_O) \\ = C_{CmbBld\_ICacheMiss}(P_O,R_O) \\ + C_{CmbBld\_MP}(P_O,R_O) \\ + C_{CmbBld\_DCacheAcc}(P_O,R_O) \quad (23)$$

$$C_{CmbBld\_ICacheMiss}(P_O,R_O) \\ = M_{L2I}(P_O,R_O) \times L_{L2} \times BF_{L2I}(P_O,R_O) \\ + M_{LLCI}(P_O,R_O) \times L_{LLC} \times BF_{LLCI}(P_O,R_O) \\ + M_{MMI}(P_O,R_O) \times L_{MM} \times BF_{MMI}(P_O,R_O) \quad (24)$$

$$C_{CmbBld\_MP}(P_O,R_O) \\ = M_{MP}(P_O,R_O) \times L_{MP} \times BF_{MP}(P_O,R_O) \quad (25)$$

$$C_{CmbBld\_DCacheAcc}(P_O,R_O) \\ = M_{L1D}(P_O,R_O) \times L_{L1} \times BF_{L2D}(P_O,R_O) \\ + M_{L2D}(P_O,R_O) \times L_{L2} \times BF_{L2D}(P_O,R_O) \\ + M_{LLCD}(P_O,R_O) \times L_{LLC} \times BF_{LLCD}(P_O,R_O) \\ + M_{MMD}(P_O,R_O) \times L_{MM} \times BF_{MMD}(P_O,R_O) \quad (26)$$

$$C_{CmbPrb\_Total}(P_O,P_{Ik},R_O,R_{Ik}) \\ = C_{CmbPrb\_ICacheMiss}(P_O,P_{Ik},R_O,R_{Ik}) \\ + C_{CmbPrb\_MP}(P_O,P_{Ik},R_O,R_{Ik}) \\ + C_{CmbPrb\_DCacheAcc}(P_O,P_{Ik},R_O,R_{Ik}) \quad (27)$$

For a combination case like Figure 5(c-2), the cost formula of the combination build phase can be created with reference to the cost formula of NLJ.

$$C_{CmbPrb\_ICacheMiss}(P_O,P_{Ik},R_O,R_{Ik}) \\ = M_{L2I}(P_O,P_{Ik},R_O,R_{Ik}) \times L_{L2} \times BF_{L2I} \\ + M_{LLCI}(P_O,P_{Ik},R_O,R_{Ik}) \times L_{LLC} \times BF_{LLCI} \\ + M_{MMI}(P_O,P_{Ik},R_O,R_{Ik}) \times L_{MM} \times BF_{MMI} \quad (28)$$

$$C_{CmbPrb\_MP}(P_O,P_{Ik},R_O,R_{Ik}) \\ = M_{MP}(P_O,P_{Ik},R_O,R_{Ik}) \times L_{MP} \times BF_{MP}(P_O,R_O,R_{Ik}) \quad (29)$$

$$C_{CmbPrb\_DCacheAcc}(P_O,P_{Ik},R_O,R_{Ik}) \\ = M_{L1D}(P_O,P_{Ik},R_O,R_{Ik}) \times L_{L1} \times BF_{L2D}(P_O,P_{Ik},R_O,R_{Ik}) \\ + M_{L2D}(P,R_O,R_{Ik}) \times L_{L2} \times BF_{L2D}(P_O,P_{Ik},R_O,R_{Ik}) \\ + M_{LLCD}(P,R_O,R_{Ik}) \times L_{LLC} \times BF_{LLCD}(P_O,P_{Ik},R_O,R_{Ik}) \\ + M_{MMD}(P,R_O,R_{Ik}) \times L_{MM} \times BF_{MM}(P_O,P_{Ik},R_O,R_{Ik}) \quad (30)$$

The aim of this study 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 measured values using the performance monitor. One of the parameters, memory latency, depends on the hardware configuration, which includes the number of CPUs, the slot position in which the main memory modules are installed, and other factors. According to J. L. Lo et al. [14], 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 Parameters Obtained for the Cost Formula

To obtain the parameters in Table II, actual measurements were made. The measurement environment is listed in Table III. We used Westmere CPUs as they have 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 the local memory, while the other is called the remote memory. In general, such a memory architecture is known as non-uniform memory access (NUMA). The latencies of the local and remote memory are different. In this study, main memory modules are installed in only one CPU to simplify the examination of measurement results. An NVM Flash SSD was used as a disk device to store the database to improve the experimental efficiency. We used the open-source MariaDB [15] as the DBMS as it supports multithreading and asynchronous I/O, can utilize the latest hardware characteristics, and supports multiple join methods. Specifically, the NLJ supported by MariaDB is a block NLJ, which is an improvement of the NLJ. However, under the conditions 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: maximum 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 (Note: storage engine's buffer cache size is scaled to be 1 TB if database size is SF 100 TB.)

The query to be evaluated and its measurement conditions are shown in Figure 6. In the SQL statement, we modified Query 3 of TPC-H for an evaluation of two-table join and extracted only join processing (Figure 6(a)). The order of joining tables is shown in Figure 6(b). This query execution plan is generated by Mariadb. The database size is scale factor (SF) 5 defined in the TPC-H specification. SF5 means that the total size of the database is 5 GB. In order to apply the proposed technology to the actual system, we used small-scale data to minimize the measurement time. The indices of the database are created on the primary keys and the foreign keys are defined in the specification of TPC-H [16].

We changed the search conditions of the query against the c_acctbal column of the outer table in order to change the selectivity of the data to be referenced (Figure 6(c)). As for NLJ, the selectivity and number of records of the inner table were changed (Figure 6(c) and (d)). The purpose of changing the selectivity is to change the total number of records accessed by the DBMS. 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 to join with the inner table. As for HJ, only the outer table was accessed in the build phase, and the number of records of the outer table was changed (Figure 6(e)). In the probe phase, only the inner table was accessed, and the number of records of the inner table was changed (Figure 6(d)). In order to accurately measure the CPU events of the build phase, the empty inner table (Figure 6(f)) was used for joining with the outer table. In order to measure CPU events without affecting DBMS behavior, the CPU events that occurred during the probe phase are measured as follows:

$$\begin{aligned}&(\textit{Number of CPU events during probe phase})\\&=(\textit{Number of CPU events during total query execution})\\&\quad-(\textit{Number of CPU events during build phase})\end{aligned}\tag{31}$$

For the combination case, the query and its measurement conditions are shown in Figure 7. This query is based on Query 3 of TPC-H. The order of joining tables is shown in Figure 7(b). This query execution plan is generated by Mariadb. The search condition and selectivity of the outer table are shown in Figure 7(c). This condition is used for modeling Figure 5(c-1). In addition, the search condition and those of the inner tables are shown in Figure 7(d). This condition for the inner table1 and the inner table2 was used for modeling Figure 5(c-2). The search condition (e) in Figure 7 was introduced to accurately measure instructions, LOAD instructions, and branch mispredictions because we found that these events were more highly affected than other events through our preliminary experiment.

The CPU performance counter data was collected using Intel® Vtune™ Amplifier XE. We refer to Levinthal (2009) [10] for a description of the content of those counters. The measured data is mainly related to the number of accesses to the cache and main memory, the state of the pipeline such as the number of stall cycles, and the number of cache hits or misses. All of the counters and the methods of preprocessing them are presented in Table B.I and B.II in Appendix B.

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 measured the I/O count and response time using systemtap and constructed the I/O cost calculation formulas by analyzing the relation between I/O and the selectivity or number of records.

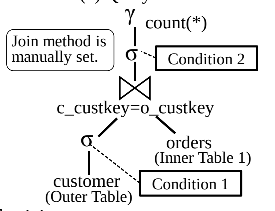
IV. Measurement Results and Cost Calculation Formulas

In this study, we investigate the relationships between selectivity, number of instructions, number of events related to memory reference, and number of branch mispredictions. For NLJ, the number of instructions and number of memory

(a) SQL

```
select count(*)
from customer, orders
where
  c_mktsegment = 'MACHINERY'  } Condition 1
  and c_acctbal > N           }
  and c_custkey = o_custkey
  and o_orderdate < date '1995-03-06';  } Condition 2
```

(b) Query Plan



(c) Selection Condition and Selectivity

N	9998	9978	9798	9200	9000
Selectivity P_O (Condition 1)	3.62×10^{-05}	4.00×10^{-04}	3.67×10^{-03}	1.45×10^{-02}	1.82×10^{-02}

(d) Condition of Inner Table

Number of Records in Inner Table	7,500,000	5,625,000	3,750,000	1,875,000

(e) Condition of Outer Table

Number of Records in Outer Table	750,000	562,500	375,000	187,500

(f) Condition of Inner Table for Measuring Build Phase

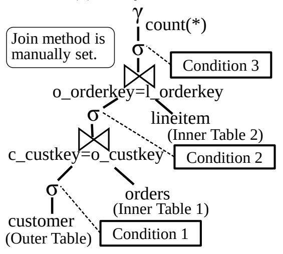
Number of Records in Inner Table	7,500,000	5,625,000	3,750,000	1,875,000
Number of Records in Outer Table	0			

Fig. 6. Target query of measurement and cost estimation for two-table join

(a) SQL

```
select count(*)
from customer, orders, lineitem
Where
  c_mktsegment = 'MACHINERY'  } Condition 1
  and c_acctbal > N           }
  and c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate < date '1995-03-06'  } Condition 2
  and l_shipdate > date '1995-03-06';  } Condition 3
```

(b) Query Plan



(c) Selection Condition and Selectivity

N	9998	9978	9798	9200	9000
Selectivity P_O (Condition 1)	3.62×10^{-5}	4.00×10^{-4}	3.67×10^{-3}	1.45×10^{-2}	1.82×10^{-2}
Selectivity P_{I1} (Condition 2)	0.482	0.482	0.482	0.482	0.482

(d) Condition of Combination Probe phase

Number of Records in Inner Table1	7,500,000	5,625,000	3,750,000	1,875,000
Number of Records in Inner Table2	37,500,000			

(e) Condition of Combination Probe phase for Instruction, LOAD Instruction and Branch Misprediction Events

Number of Records in Inner Table1	7,500,000	5,625,000	3,750,000	1,875,000
Number of Records in Inner Table2	52,500,000			
	37,500,000			
	22,500,000			

Fig. 7. Target query of measurement and cost estimation for three-table join

references are expected to increase because the number of records accessed by the DBMS increases in proportion to the increase in selectivity. Based on the assumptions, we now analyze the measurement results and create formulas using linear regression. For HJ, all of the records of the outer table and inner table were accessed in both the build phase and probe phase. The cost formulas were assumed to not have selectivity as a variable; we analyzed the measurement results based on this assumption. For the combination of HJ and NLJ, the combination build phase was considered to be the same as the build phase of HJ. We investigated the relationships between the number of selected records from the outer table, number of records in the inner tables, and number of CPU events.

The CPU cost calculation formulas were obtained through the following steps. First, the number of instructions, references of each cache memory, and main memory and branch mispredictions were analyzed using regression analysis, and the regression models were created. In addition, the relationship between the sum of the product of the references to each memory and its latency, and $C_{ICacheMiss}$ (10) and $C_{DCacheAcc}$ (13) were modeled. Here, C_{MP} (11) was 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 from [10] [17]. The number of disk I/O was modeled using the measured I/O access count and I/O response time. Finally, the cost calculation formulas were evaluated from the viewpoint of the accuracy of intersection of the two join methods (X_{cross} in Figure 2) with the conventional method.

Figure 8(1) shows the relationship between the number of records the DBMS accessed and load instructions. Figure 8(7) shows the relationship between the total number of accessed records and number of instructions. The number of records is the product of the number of outer table records, number of inner table records, and selectivity. The dotted line is the linear regression line, and its slope and intercept are listed 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 linear regression model is highly accurate. The slope and intercept were used to create the cost calculation model. Figures 8(2) and (8) show the relationship between the number of instructions executed by the DBMS and the number of L1 cache hits. Figures 8(3)–(6) and (9)–(12) show the relationships between the number of accesses to L2, LLC, and main memory, and the number of cache misses of the upper-level cache. These relationships 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 work, a two-CPU server was used and the LLC and main memory were connected to each CPU. The LLC and main memory of the CPU on which DBMS 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 are no references to the remote main memory because the main memory is connected to only one CPU in our experimental environment. Figure 8(13) shows the relationship between the number of records accessed for the join operation and the branch misprediction cycles C_{MP}. Figure 8(14) shows the relationship between the product of the number of instruction accesses and latency, and the L1I miss cycles (miss penalty), $C_{ICacheMiss}$. Figure 8(15) shows the relationship between the products of the number of data accesses and latency, and the data cache and main memory access, $C_{DCacheAcc}$. Each graph can also 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.

Figures 9(a1)–(a15), (b1)–(b15) and Figures 10(1)–(15) show the tendency of instructions, cache or main memory accesses, branch misprediction cycles, instruction cache miss cycles, and data cache access cycles. These events tend to be similar to those of the NLJ. The dotted line is the linear regression line, and its slope and intercept are shown in Table IV and Table V. The coefficient of determination (R^2) is near 1 and the P value on the F test is less than 0.05.

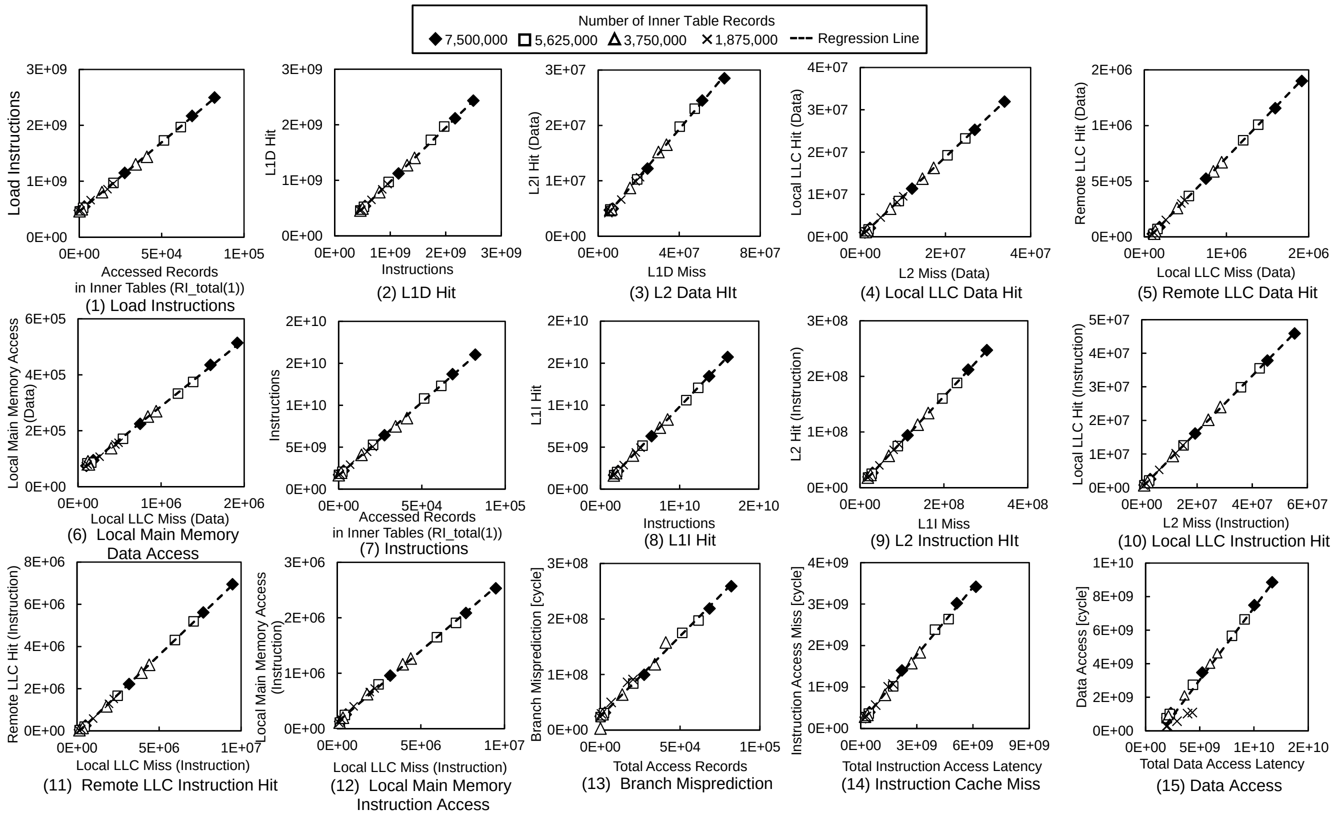


Fig. 8. CPU event count on executing NLJ

Therefore, the linear regression model is highly accurate. The slope and intercept are used for creating the cost calculation model.

In particular, the slope of the regression line in Figures 8(2)–(5) and (9)–(11); Figures 9(a2)–(a5), (a9)–(a11), (b2)–(b5), and (b9)–(b11); and Figures 10(2)–(5) and (9)–(11) represents the cache hit rate because the definition of cache hit rate is the quotient of the number of cache hits and number of cache references, and the upper-level cache miss becomes the lower-level cache reference.

In this study, the number of cache hits is chosen as an explanatory variable, as shown in Figure 11(a), which is the same graph as that in Figure 8(8). In general, the cache hit ratio is more often used for modeling CPU memory access than the number of cache hits. However, the cache hit ratio graph (Figure 11(b)) has a hyperbolic shape. The CPU cost calculation function should be simple in order to apply a simple formula to the actual DBMS. In addition, the reason why the cache hit ratio graph has a hyperbolic shape is explained by the following expressions (32) and (33).

The regression line of Figure 11(a) is

$$M_{L1I} = A \times I + B, \tag{32}$$

where A and B are the slope and intercept of a linear regression on the two table join in Figure 6, respectively. The cache hit ratio is obtained by dividing the number of cache hits by that of instructions. Therefore, the cache hit ratio (33) is obtained by dividing both sides of (32) by I.

$$(L1I\ Hit\ Ratio) = A + \frac{B}{I}, \tag{33}$$

In order to apply the two-table join calculation model to three or more tables, it is necessary to estimate the total number of accessed records in the inner tables (Figure 4(a), Figure 5 (c-1)). As shown in Figure 12, the number of accessed records in inner table1 is $R_{I0} \times P_{I0} \times rsk_0$ where rsk_0 is ratio of R_{I1} to R_{I0} (34). If $R_{I1} < R_{I0}$, then $rsk_0 = 1$ because the number of accessed records in inner table1 is as large as the references from the outer table. The number of references from inner table1 to inner table2 is $R_{I0} \times P_{I0} \times rsk_0 \times P_{I1} \times rsk_1$. Therefore, the total number of accessed records in the inner tables is $(R_{I0} \times P_{I0} \times rsk_0) + (R_{I0} \times P_{I0} \times rsk_0 \times P_{I1} \times rsk_1)$. Based on the above, we introduce rsk and $R_{I\_total}(n)$, which are written as (34) and (35).

$$rsk_i = \begin{cases} \dfrac{R_{Ii}}{R_{I(i-1)}} & R_{Ii} \geq R_{I(i-1)} \text{ where } R_{I0} = R_O \\ 1 & R_{Ii} < R_{I(i-1)} \end{cases} \tag{34}$$

$$R_{I\_total(n)} = R_O \times \sum_{j=1}^{n} \prod_{i=0}^{j-1} (rsk_i \times P_{Ii}) \tag{35}$$

where n of $R_{I\_total}(n)$ means the number of inner tables to join.

In the combination probe phase, multiple inner tables are traversed with the key of the records in the hash table.

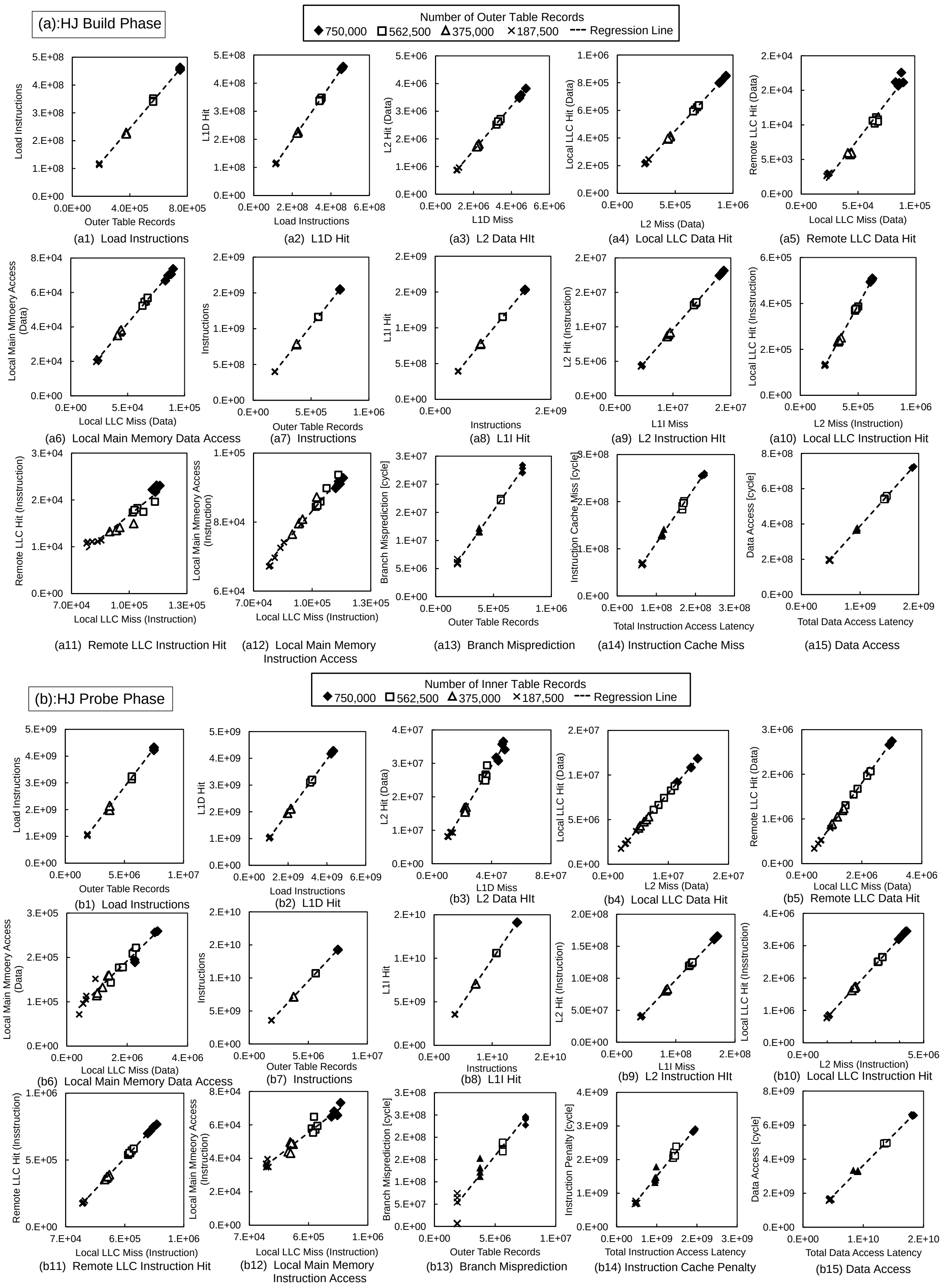


Fig. 9. CPU event count on executing HJ: (a) build phase, (b) probe phase)

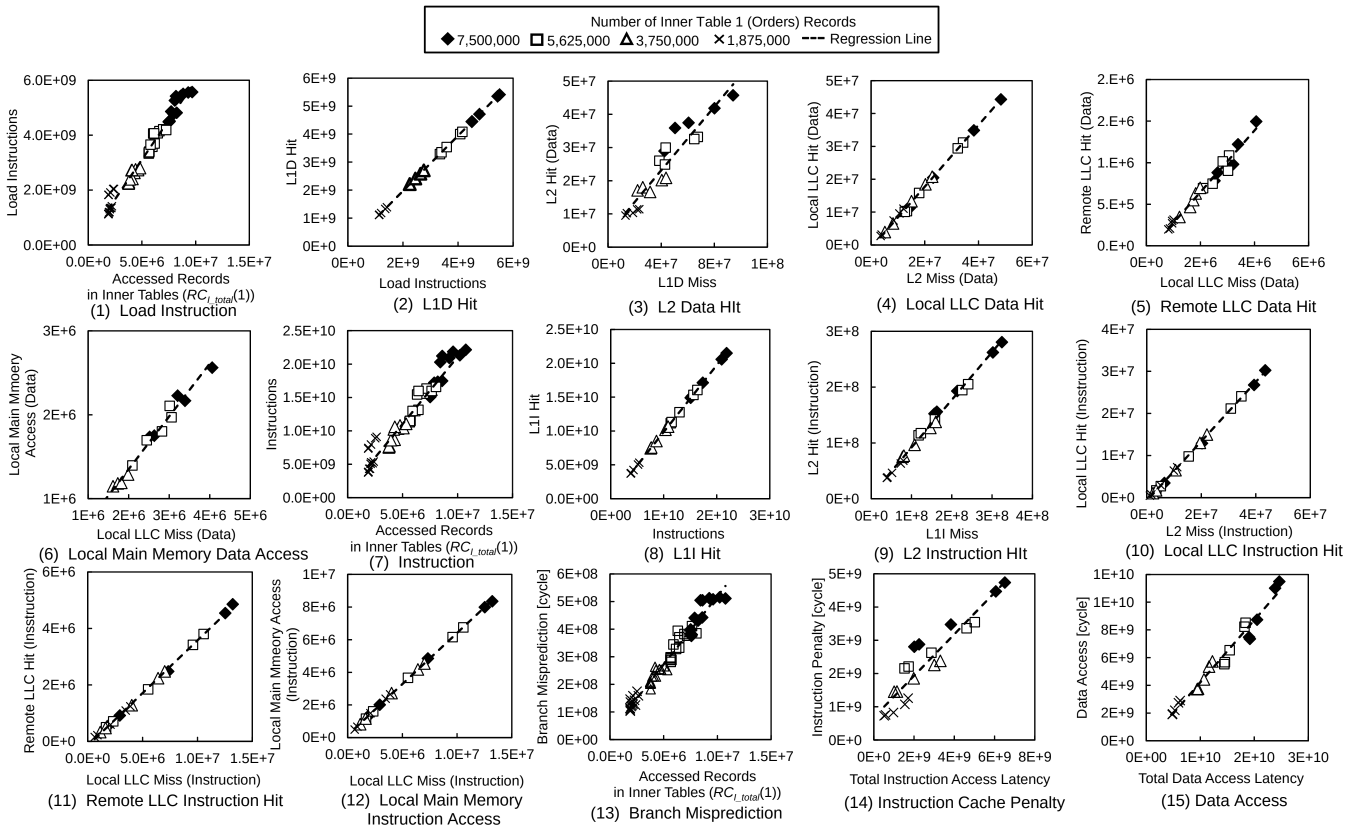


Fig. 10. CPU event count on combination probe phase

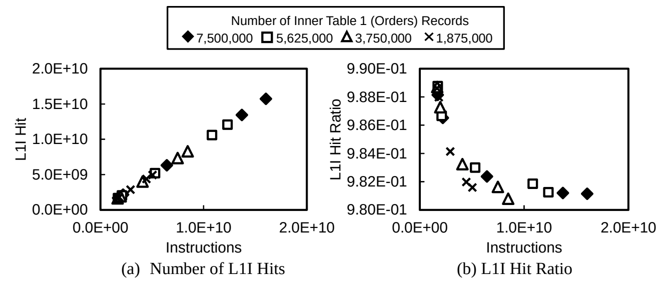


Fig. 11. Problem of modeling cache hit ratio

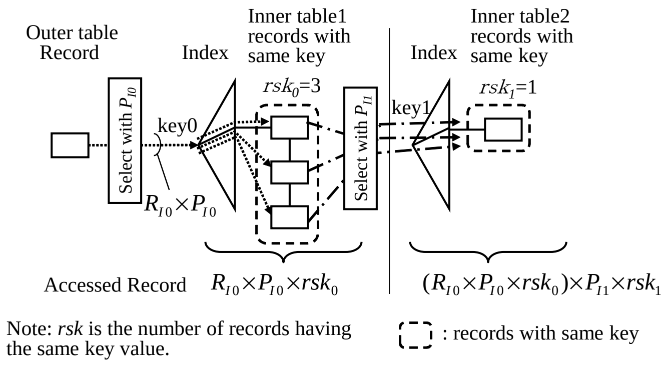

Outer table Record
Index
Inner table1 records with same key
Select with P_{I0}
Index
Inner table2 records with same key
rsk_0=3
rsk_1=1
key0
key1
Select with P_{I1}
$R_{I0}\times P_{I0}$
Accessed Record
$R_{I0}\times P_{I0}\times rsk_0$
$(R_{I0}\times P_{I0}\times rsk_0)\times P_{I1}\times rsk_1$
Note: rsk is the number of records having the same key value.
: records with same key

Fig. 12. Simple example of number of records having the same key value (rsk)

The number of instructions, LOAD instructions, and branch mispredictions are proportional to the number of inner table records accessed by the DBMS. However, the number of these events also depends on the number of entries in the hash table. It is necessary to obtain the sum of these two kinds of records with different properties. In general, the height of the index is approximately 3 to 4 as more than 100 records are registered in each node of the B+ tree. When the cost calculation equations are a function of only the accessed records in the inner tables, the traversing records among nodes and inside nodes can be considered as constant and omitted from the cost calculation model. However, in order to consider the scan of the hash table at the same time, it is necessary to consider both index height and records having the same key. Therefore, in the combination probe phase, $RC_{I\_total}(n)$ was introduced to construct a cost calculation formula. $RC_{I\_total}(n)$ is given by (36) as follows:

$$RC_{I\_total(n)} = R_O \times \left\{ \sum_{j=1}^{n} \prod_{i=0}^{j-1} (rsk_i \times P_{Ii}) \times \left(log_t(R_{Ii}) + \frac{rsk_i + 1}{2} \right) + 1 \right\} \quad (36)$$

where t is the number of entries stored in an index page. t is 100 because the B+ tree index stores more than 100 records in an index page.

The number of instructions, LOAD instructions, and branch mispredictions in the NLJ and combination probe phase are proportional to the number of referenced records in the inner tables (Figure 8(1)(7)(13) and Figure 10(1)(7)(13)).

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

$$I = A1 \times R + B1 \tag{37}$$

$$M_{L1I} = A2 \times I + B2 \tag{38}$$

$$M_{L2I} = A3 \times (I - M_{L1I}) + B3 \tag{39}$$

$$M_{LLLCI} = A4 \times (I - M_{L1I} - M_{L2I}) + B4 \tag{40}$$

$$M_{RLLCI} = A5 \times (I - M_{L1I} - M_{L2I} - M_{LLLCI}) + B5 \tag{41}$$

$$M_{LMMI} = A6 \times (I - M_{L1I} - M_{L2I} - M_{LLLCI}) + B6 \tag{42}$$

$$I_{Load} = A7 \times R + B7 \tag{43}$$

$$M_{L1D} = A8 \times I_{Load} + B8 \tag{44}$$

$$M_{L2D} = A9 \times (I - M_{L1D}) + B9 \tag{45}$$

$$M_{LLLCD} = A10 \times (I - M_{L1D} - M_{L2D}) + B10 \tag{46}$$

$$M_{RLLCD} = A11 \times (I - M_{L1D} - M_{L2D} - M_{LLLCD}) + B11 \tag{47}$$

$$M_{LMMD} = A12 \times (I - M_{L1D} - M_{L2D} - M_{LLLCD}) + B12 \tag{48}$$

$$\begin{aligned} C_{ICacheMiss} = A13 \times (&M_{L2I} \times L_{L2} + M_{LLLCI} \times L_{LLLC} \\ &+ M_{RLLCI} \times L_{RLLC} + M_{LMMI} \times L_{LMM}) + B13 \end{aligned} \tag{49}$$

$$\begin{aligned} C_{DCacheAcc} = A14 \times (&M_{L1D} \times L_{L1} + M_{L2D} \times L_{L2} \\ &+ M_{LLLCD} \times L_{LLLC} + M_{RLLCD} \times L_{RLLC} \\ &+ M_{LMMD} \times L_{LMM}) + B14 \end{aligned} \tag{50}$$

$$C_{MP} = A15 \times R + B15 \tag{51}$$

where

$$R = \begin{cases} R_{I\_total(n)} & \text{NLJ} \\ R_O & \text{HJ build phase and combination build phase} \\ R_I & \text{HJ probe phase} \\ RC_{I\_total(n)} & \text{Combination probe phase} \end{cases}$$

The cost calculation formulas ((14), (37)–(51)) can become the following single formula by focusing on R ((52), (53), (54)). This formula suggests that our approach means estimating an accurate unit CPU cost per accessed record.

$$C_{Total} = \alpha \times R + \beta \tag{52}$$

where

$$\begin{aligned} \alpha = {} & A1 \times A13 \times (L_{L2} \times A3 \times (1 - A2) \\ & + L_{LLLC} \times A4 \times (1 - A3 - A2 + A2 \times A3) \\ & + L_{RLLC} \times A5 \times (1 - A2 - A3 + A2 \times A3 - A4 + A3 \times A4 \\ & + A2 \times A4 - A2 \times A3 \times A4) \\ & + L_{LMM} \times A6 \times (1 - A2 - A3 + A2 \times A3 - A4 + A3 \times A4 \\ & + A2 \times A4 - A2 \times A3 \times A4)) \\ & + A7 \times A14 \times (L_{L1} \times A8 + A14 \times L_{L2} \times (A9 - A8 \times A9) \\ & + L_{LLLC} \times A10 \times (1 - A9 - A8 + A8 \times A9) \\ & + L_{RLLC} \times A11 \times (1 - A8 - A9 + A8 \times A9 - A10 + A9 \times A10 \\ & + A8 \times A10 - A8 \times A9 \times A10) \\ & + L_{LMM} \times A12 \times (1 - A8 - A9 + A8 \times A9 - A10 + A9 \times A10 \\ & + A8 \times A10 - A8 \times A9 \times A10)) + A15 \end{aligned} \tag{53}$$

$$\begin{aligned} \beta = {} & A13 \times (L_{L2} \times (-A3 \times B2 + B3) \\ & + L_{LLLC} \times (-A4 \times B2 + A3 \times A4 \times B2 - A4 \times B3 + B4) \\ & + L_{RLLC} \times (-A5 \times B2 + A3 \times A5 \times B2 - A5 \times B3 \\ & + A4 \times A5 \times B2 - A3 \times A4 \times A5 \times B2 + A4 \times A5 \times B3 \\ & - A5 \times B4 + B5) \\ & + L_{LMM} \times (-A6 \times B2 + A3 \times A6 \times B2 - A6 \times B3 + A4 \\ & \times A6 \times B2 - A3 \times A4 \times A6 \times B2 + A4 \times A6 \times B3 - A6 \times B4 \\ & + B5)) + A14 \times (L_{L1} \times B8 + L_{L2} \times (-A9 \times B8 + B9) \\ & + L_{LLLC} \times (-A10 \times B8 + A9 \times A10 \times B8 - A10 \times B9 + B10) \\ & + L_{RLLC} \times (-A11 \times B8 + A9 \times A11 \times B8 - A11 \times B9 \\ & + A10 \times A11 \times B8 - A9 \times A10 \times A11 \times B8 + A10 \times A11 \times B9 \\ & - A11 \times B10 + B11) \\ & + L_{LMM} \times (-A12 \times B8 + A9 \times A12 \times B8 - A12 \times B9 \\ & + A10 \times A12 \times B8 - A9 \times A10 \times A12 \times B8 + A10 \times A12 \times B9 \\ & - A12 \times B10 + B11)) + B13 + B14 + B15 \end{aligned} \tag{54}$$

Table IV lists the definitions of the parameters given in (37)–(51) for NLJ and HJ. In the case of NLJ, the calculation formula of the number of disk I/Os was created using the regression line shown in Figure 13(a). The measured I/O response time (*io_responcetime*) was 154 μs. The I/O cost of NLJ is as follows:

$$io\_cost = (A16 \times R_O \times R_{I1} \times P_O + B16) \times io\_responce\_time \tag{55}$$

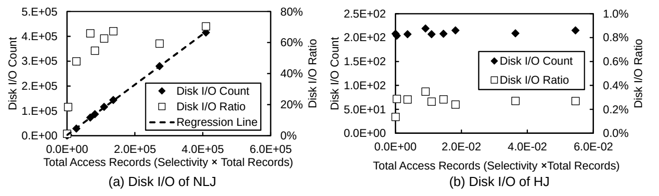


Fig. 13. Number of disk I/O and disk I/O processing time during NLJ and HJ

However, in the case of HJ, the ratio of the processing time of disk I/O and the query execution time of HJ was less than 1% in Figure 13(b). The cost calculation formula is composed of only the CPU cost and disk I/O cost. In order to apply in-memory databases (Figure 1(b)), it is sufficient to change the disk I/O latency to the latency of the memory based disk.

In the combination probe phase, the calculation formula for the number of disk I/Os was created using the multiple regression line shown in Figure 14. The I/O cost of combination probe phase is as follows:

$$io\_cost = (A17 \times R_{I1} \times P_{I1} + A18 \times R_{I1} \times \sum_{i=1}^{n} rsk_i + B17) \times io\_responce\_time \tag{56}$$

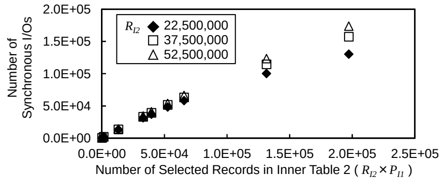


Fig. 14. Number of synchronous disk I/O count during combination probe phase

The regression models for the combination join are also expressed by the same equations, (37)–(51), as the two-table NLJ and HJ. Table V lists the definitions of the parameters.

TABLE IV. SLOPE AND INTERCEPT OF THE REGRESSION MODELS

Type	Slope (Regression Coefficient)		Intercept (Regression Constant)		R^2	P value on F test	Reference
NLJ	A1	1.745×10^{5}	B1	1.64×10^{9}	9.99×10^{-1}	1.30×10^{-29}	Figure 8(1)
	A2	9.80×10^{-1}	B2	1.26×10^{7}	1.00	2.18×10^{-58}	Figure 8(2)
	A3	8.08×10^{-1}	B3	2.68×10^{6}	1.00	2.91×10^{-40}	Figure 8(3)
	A4	8.32×10^{-1}	B4	5.23×10^{4}	1.00	3.93×10^{-39}	Figure 8(4)
	A5	7.43×10^{-1}	B5	-1.18×10^{5}	1.00	3.77×10^{-34}	Figure 8(5)
	A6	2.58×10^{-1}	B6	1.18×10^{5}	9.98×10^{-1}	7.05×10^{-26}	Figure 8(6)
	A7	2.46×10^{4}	B7	4.63×10^{8}	9.99×10^{-1}	5.97×10^{-31}	Figure 8(7)
	A8	9.72×10^{-1}	B8	7.05×10^{6}	1.00	3.85×10^{-53}	Figure 8(8)
	A9	4.34×10^{-1}	B9	1.95×10^{6}	9.99×10^{-1}	5.17×10^{-28}	Figure 8(9)
	A10	9.44×10^{-1}	B10	-2.87×10^{4}	1.00	2.06×10^{-44}	Figure 810)
	A11	7.61×10^{-1}	B11	-4.84×10^{4}	1.00	2.80×10^{-35}	Figure 8(11)
	A12	2.39×10^{-1}	B12	4.84×10^{4}	9.98×10^{-1}	3.10×10^{-26}	Figure 8(12)
	A13	5.45×10^{-1}	B13	1.41×10^{8}	9.98×10^{-1}	1.21×10^{-25}	Figure 8(13)
	A14	8.59×10^{-1}	B14	-1.25×10^{9}	9.67×10^{-1}	9.00×10^{-15}	Figure 8(14)
	A15	2.90×10^{3}	B15	2.40×10^{7}	9.90×10^{-1}	1.35×10^{-19}	Figure 8(15)
HJ Build	A1	2.05×10^{3}	B1	1.58×10^{7}	1.00	4.21×10^{-40}	Figure 9(a1)
	A2	9.88×10^{-1}	B2	2.53×10^{5}	1.00	2.19×10^{-61}	Figure 9(a2)
	A3	9.71×10^{-1}	B3	-7.48×10^{4}	1.00	1.79×10^{-49}	Figure 9(a3)
	A4	9.19×10^{-1}	B4	-6.57×10^{4}	9.99×10^{-1}	7.76×10^{-31}	Figure 9(a4)
	A5	3.32×10^{-1}	B5	-1.64×10^{4}	9.29×10^{-1}	8.38×10^{-12}	Figure 9(a5)
	A6	6.68×10^{-1}	B6	1.64×10^{4}	9.82×10^{-1}	4.49×10^{-17}	Figure 9(a6)
	A7	6.10×10^{2}	B7	2.85×10^{5}	9.99×10^{-1}	4.46×10^{-30}	Figure 9(a7)
	A8	9.90×10^{-1}	B8	-1.89×10^{3}	1.00	1.05×10^{-57}	Figure 9(a8)
	A9	8.03×10^{-1}	B9	-2.39×10^{4}	1.00	6.00×10^{-33}	Figure 9(a9)
	A10	9.04×10^{-1}	B10	1.58×10^{2}	1.00	8.94×10^{-46}	Figure 9(a10)
	A11	2.13×10^{-1}	B11	-2.74×10^{3}	9.80×10^{-1}	1.13×10^{-16}	Figure 9(a11)
	A12	7.87×10^{-1}	B12	2.74×10^{3}	9.98×10^{-1}	8.16×10^{-27}	Figure 9(a12)
	A13	1.20	B13	-8.24×10^{6}	9.98×10^{-1}	2.14×10^{-25}	Figure 9(a13)
	A14	3.69×10^{-1}	B14	2.02×10^{7}	1.00	6.83×10^{-32}	Figure 9(a14)
	A15	2.94×10^{1}	B15	6.41×10^{5}	9.97×10^{-1}	2.86×10^{-24}	Figure 9(a15)
HJ Probe	A1	1.90×10^{3}	B1	2.33×10^{7}	1.00	3.46×10^{-46}	Figure 9(b1)
	A2	9.88×10^{-1}	B2	3.88×10^{5}	1.00	3.69×10^{-62}	Figure 9(b2)
	A3	9.75×10^{-1}	B3	-1.76×10^{4}	1.00	6.81×10^{-52}	Figure 9(b3)
	A4	8.13×10^{-1}	B4	-2.44×10^{4}	1.00	1.12×10^{-41}	Figure 9(b4)
	A5	9.46×10^{-1}	B5	-2.44×10^{4}	1.00	8.30×10^{-36}	Figure 9(b5)
	A6	5.45×10^{-2}	B6	2.44×10^{4}	9.56×10^{-1}	1.15×10^{-13}	Figure 9(b6)
	A7	5.76×10^{2}	B7	-3.57×10^{7}	9.99×10^{-1}	6.14×10^{-27}	Figure 9(b7)
	A8	9.89×10^{-1}	B8	-3.89×10^{5}	1.00	6.68×10^{-54}	Figure 9 (b8)
	A9	7.29×10^{-1}	B9	1.58×10^{5}	9.88×10^{-1}	7.30×10^{-19}	Figure 9(b9)
	A10	7.95×10^{-1}	B10	2.81×10^{4}	1.00	5.98×10^{-33}	Figure 9 (b10)
	A11	9.34×10^{-1}	B11	-6.04×10^{4}	1.00	7.79×10^{-34}	Figure 9 (b11)
	A12	6.60×10^{-2}	B12	6.04×10^{4}	9.52×10^{-1}	2.69×10^{-13}	Figure 9(b12)
	A13	1.48	B13	2.87×10^{07}	9.87×10^{-1}	1.40×10^{-18}	Figure 9(b13)
	A14	3.58×10^{-1}	B14	6.61×10^{7}	9.98×10^{-1}	1.75×10^{-25}	Figure 9(b14)
	A15	3.45×10^{1}	B15	-1.39×10^{7}	9.35×10^{-1}	3.77×10^{-12}	Figure 9(b15)
NLJ (I/O)	A16	1.02	B16	2.52×10^{3}	1.00	1.85×10^{-15}	Figure 13(a)
HJ (I/O)	A16	0.000	B16	0.000	N/A	N/A	N/A

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* and *lineitem*. In addition, in order to evaluate the join of more

TABLE V. SLOPE AND INTERCEPT OF THE REGRESSION MODELS IN COMBINATION PROBE PHASE

Type	Slope (Regression Coefficient)		Intercept (Regression Constant)		R^2	P value on F test	Reference
Probe	A1	2.01×10^{3}	B1	9.80×10^{8}	9.42×10^{-1}	1.63×10^{-37}	Figure 10(1)
	A2	9.86×10^{-1}	B16	1.79×10^{7}	1.00	1.11×10^{-44}	Figure 10(2)
	A3	8.53×10^{-1}	B3	6.58×10^{6}	9.94×10^{-1}	2.79×10^{-21}	Figure 10(3)
	A4	7.06×10^{-1}	B4	-6.80×10^{5}	9.99×10^{-1}	3.12×10^{-29}	Figure 10(4)
	A5	3.74×10^{-1}	B5	-1.58×10^{5}	9.99×10^{-1}	3.51×10^{-28}	Figure 10(5)
	A6	6.26×10^{-1}	B6	1.58×10^{5}	1.00	3.21×10^{-32}	Figure 10(6)
	A7	5.76×10^{2}	B7	2.31×10^{8}	9.75×10^{-1}	2.78×10^{-48}	Figure 10(7)
	A8	9.86×10^{-1}	B8	1.83×10^{6}	1.00	1.75×10^{-41}	Figure 10(8)
	A9	4.79×10^{-1}	B9	3.97×10^{6}	9.00×10^{-1}	2.03×10^{-10}	Figure 10(9)
	A10	9.36×10^{-1}	B10	-9.55×10^{5}	9.97×10^{-1}	7.85×10^{-25}	Figure 1010)
	A11	3.70×10^{-1}	B11	-9.15×10^{4}	9.75×10^{-1}	6.97×10^{-16}	Figure 10(11)
	A12	6.30×10^{-1}	B12	9.15×10^{4}	9.91×10^{-1}	5.74×10^{-12}	Figure 10(12)
	A13	6.25×10^{-1}	B13	6.85×10^{8}	8.75×10^{-1}	1.51×10^{-9}	Figure 10(13)
	A14	4.49×10^{-1}	B14	-2.66×10^{8}	9.72×10^{-1}	2.16×10^{-15}	Figure 10(14)
	A15	4.93×10^{1}	B15	2.62×10^{7}	9.70×10^{-1}	1.03×10^{-45}	Figure 10(15)
Probe (I/O)	A17	7.84×10^{-1}	B17	-3.24×10^{-4}	9.77×10^{-1}	7.43×10^{-21}	N/A
	A18	2.43×10^{3}					N/A

than two tables, the following combinations were chosen: (*customer*, *orders*, *lineitem*), (*supplier*, *lineitem*, *part*, *orders*, *customer*), and (*part*, *lineitem*, *supplier*, *orders*, *customer*). In the five-table join case, the STRAIGHT_JOIN query hint was used to keep the join order of tables. Detailed measurement conditions are shown in Appendix A. The parameter setting of the cost calculation formulas was generated from the measurement values when joining *customer* and *orders*, whose size is SF5. For the join cases of three or more tables, the measurement values under joining *customer*, *orders*, and *lineitem* were used. The I/O processing time was added to allow comparison with the query execution time. The proposed cost calculation method was compared with the measured query execution time and conventional method (2)(3). The conventional method is expressed as the sum of the CPU cost and the I/O cost as mentioned in Section I. Each cost is calculated as the product of the number of records and the manually defined processing unit cost of CPU or I/O. The conventional method and proposed method followed the execution plan generated by MariaDB. In our measurement environment, the HJ execution plan for joining more than two tables is the combination case (Figure 5). We evaluated 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 and inner tables were 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 was 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, the proposed method was able to find the intersection point with an accuracy of one significant figure or better compared to the conventional method (Figure 15 and Figure 16). The improvement ratios of the proposed method and conventional method are shown in Table VI. The second and third rows indicate the difference between the intersection point (selectivity) of the measured result and that of the conventional method or proposed method. The improvement ratio was obtained by dividing the difference between the intersection selectivity of the conventional method

TABLE VI. IMPROVEMENT RATIO FOR ESTIMATING INTERSECTION OF NLJ AND HJ

Join tables	C-O	P-L	S-L	C-O-L	S-L-P-O-C	P-L-S-O-C
Conventional method	1.5×10^{-2}	3.5×10^{-3}	2.2×10^{-3}	7.5×10^{-3}	8.9×10^{-2}	2.2×10^{-5}
Proposed method	1.3×10^{-4}	3.2×10^{-4}	1.9×10^{-4}	8.8×10^{-5}	3.0×10^{-4}	2.2×10^{-7}
Improvement ratio	99%	91%	91%	99%	97%	99%

Note: C: Customer, O: Orders, L: Lineitem, P: Part, S: Supplier

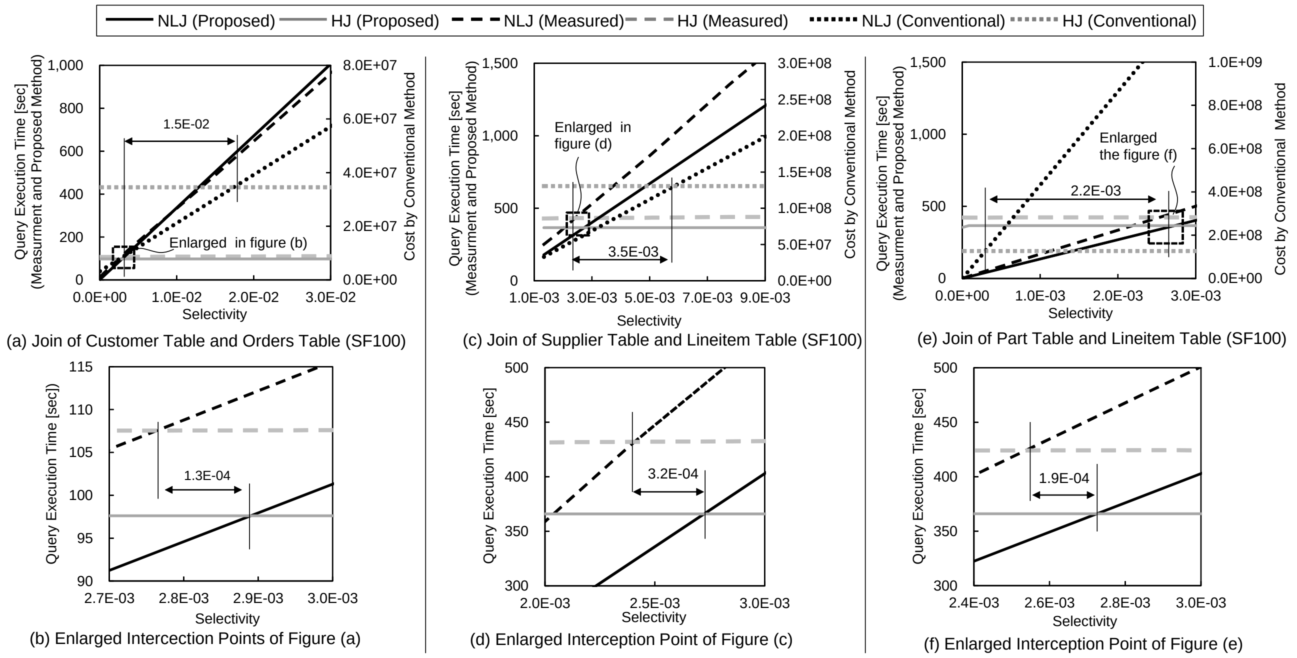


Fig. 15. Cost Comparison of measured, proposed cost model, and conventional cost model results for joining two tables

and that of the proposed method by that of the conventional method. Table VI shows that the proposed method improved the accuracy of selecting the proper join method by 90% or more.

V. Discussion

In the acquisition of measurement data for constructing the cost calculation formula, because the type of counters that the hardware monitor can collect at one time is limited to four, it is necessary to perform measurements several times to obtain an accurate measurement of 40 events. Therefore, a certain amount of time must be allocated for measurement. For example, it took approximately 5 h and 30 min to perform the measurements in this study. From the perspective of allocating time for measurement, and given the fact that the CPU cost calculation formula does not need to be changed unless there is a change in hardware configuration or DBMS join operation codes, it is appropriate to create the proposed CPU cost calculation formula when integrating or updating a system. With regard to the use of the cost calculation formula, the proposed CPU cost formula was used in the optimization process to be executed before executing a query. The CPU cost of executing the query was calculated from the number of records to be searched. As shown in references [18] [19], in a general DBMS, the histograms representing the relationship between the attribute value and appearance frequency are automatically acquired when inserting or updating records. From the histogram and condition of the clause of the query, it is possible to estimate the number of records accessed by the DBMS. In this way, the CPU costs can be calculated with only the data already acquired by the DBMS; hence, the costs can be calculated by the cost calculation formula before query execution.

The combination join case modeled in this study was the two or more tables join case, as shown in Figure 5. Theoretically, there is a case in which HJ is assigned after NLJ. In the case where HJ was executed after NLJ (Figure 17), the cost model (d-1) was the same as (a), and the cost model (d-3) was the same as (b-2). The cost model of (d-2) was required because building the hash table from temporary table X was not covered in the other case. Most of the join cases seem to be classified as in Figures 4, 5, and 17, and creating the cost models (a), (b-1), (b-2), (c-2), and (d-2) can support most of the join cases. However, the NLJ–HJ case cannot be implemented in our measurement environment. This problem can be solved by using a different DBMS.

We proposed a cost calculation method for an in-memory DBMS using a disk-based DBMS. The calculation formulas were created using the data measured by the CPU-embedded performance monitor. The results reveal that the proposed method can estimate the intersection point of the join methods more accurately than the conventional method. We used TPC-

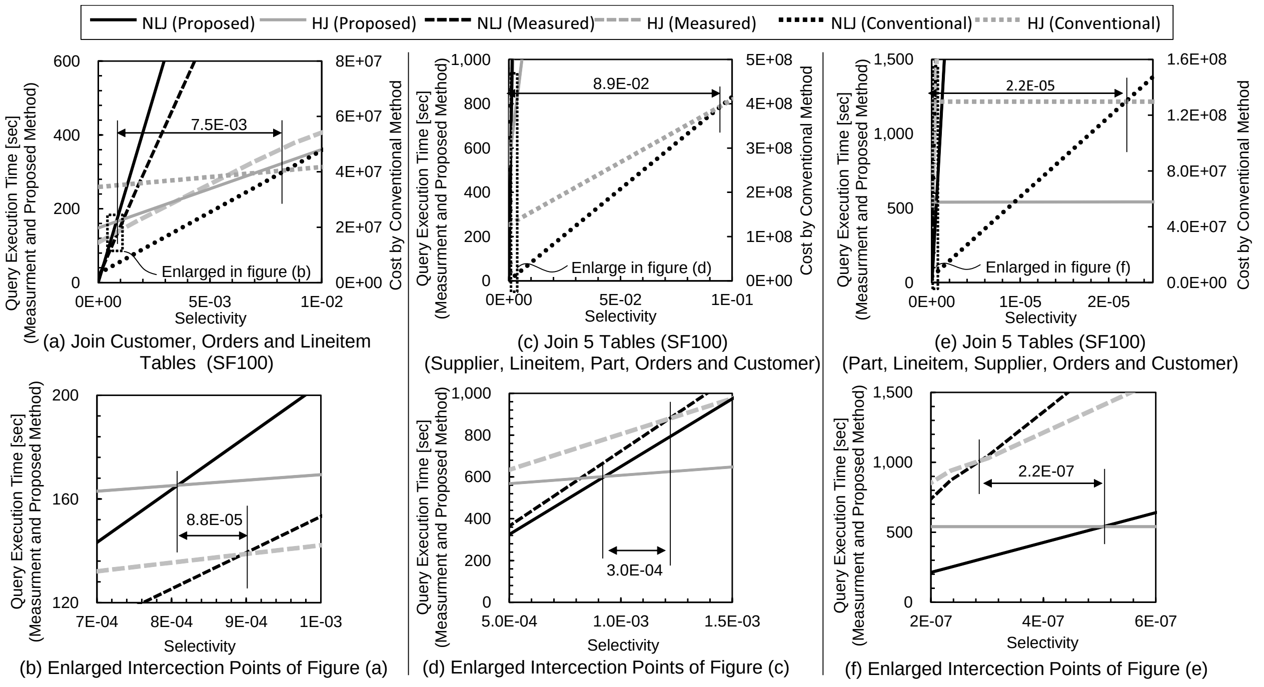


Fig. 16. Cost Comparison of measured, proposed cost model, and conventional cost model results for joining three or more tables

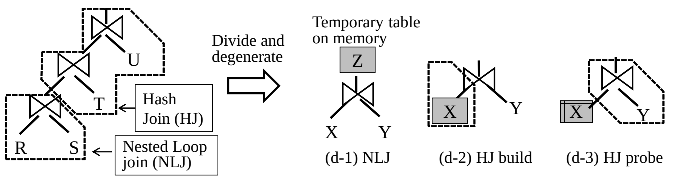


Fig. 17. NLJ after HJ Case

H for measuring CPU activities. TPC-H has the advantage of making it easy to analyze the evaluation results because the data distribution is uniform. However, the actual data is skewed in terms of the distribution of keys. The premise of the technique proposed in this study is the accuracy of selectivity, i.e., even if the distribution of data varies, if the selectivity is the same, then the same measurement results are obtained. Because a general DBMS acquires attribute values and their distribution in a database is in the form of a histogram when loading data to the database, the prerequisites for applying the proposed technique are 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.

As 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 complex queries. Although we have focused on the operation of all CPU cycles, it is necessary for practical use to simplify the model by omitting some parameters. For the collection of statistical data, it is conceivable that actual measurements can be performed at the time of initial installation and parameter setting. However, when the DBMS code is modified, it is difficult to change in real time; hence, a separate complementary technology is required. As a breakthrough measure, it is possible to reduce both the amount of data to be verified and the measurement points.

In this study, only the cost model for join operations was proposed. However, the query operation includes not only join but also filter, group-by, and sorting operations. These operations are difficult to execute by using only a query. However, part of the query can be extracted by taking the difference between queries, similar to our approach for modeling the HJ probe phase (31).

VI. Related Work

Evaluation of the CPU performance using the performance monitor for behavior analysis of a DBMS has long been conducted. In particular, in the evaluation of the benchmark TPC-D for decision support systems, the L1 miss and processing delay owing to the L2 cache occupy a large part of the CPI components, and are important in terms of performance. However, these are only used for bottleneck analysis [20].

The query execution cost calculation approaches include the white-box analysis [8] and the black-box analysis [9]. In the white-box analytic approach, the cost calculation model for a single server is composed of CPU cost and I/O cost. Based on these approaches, there are some cost calculation methods. One is the product of a unit cost and the number of accessed records [5] [6]. Another is to estimate cost from the execution time of several evaluation queries [8]. With the black box-model analysis approach, a multiple regression is performed using parameters that the DBMS user can refer easily, such as the cardinality of tables. Our approach combines both characteristics.

From a different viewpoint, there exist the macro-level and micro-level approaches. The macro-level approach is suitable for a heterogeneous DBMS system because it is composed of different DBMSs (open source or commercial DBMSs) and cost is calculated based on the processing time of commonly executable queries. Our approach is a micro-level one. It is created from measurement results of CPU events while executing a query. The micro-level approach can create an accurate model by considering the CPU operation, but it cannot be applied to different DBMS.

Our cost calculation model uses the statistic information of the CPU measured under a static environment. However, multiple applications are executed on a real production system. The cost model of a multiple-application environment is based on multiple regression models and it uses sample-query execution time and statistical calibration methods [21] [22]. Applying these methods to our approach will help achieve a more accurate model.

Another study on the micro-level approach is the method that applies a CPI measurement and focuses on a memory reference for cost calculations (5) of an in-memory database [23] [24]. This research targeted a DBMS that use the load/store type memory access (Figure 1(c)). In this work, the number of cache hits or main memory accesses was predicted from the data access pattern of the database, and the cost was calculated as the product of the number of cache hits or main memory accesses and the memory latency. The modeling of *CPI0*, which is the state where all data exist 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 were on the L1 cache, which is the definition of *CPI0*, using methods such as a CPU-embedded performance monitor.

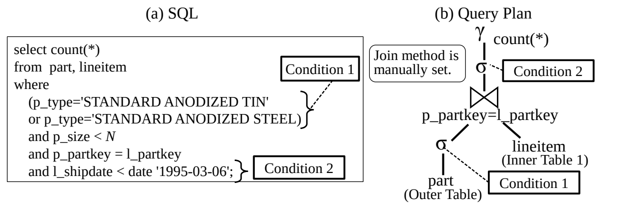
In our research, the performance of queries was considered as a function of selectivity. Kester *et al.* [25] used not only selectivity but also the concurrency of queries in the execution to calculate the query execution cost. When many queries are executed simultaneously in a cloud computing system, hardware resources (e.g., memory bandwidth, disk bandwidth, etc.) will become scarce. In this case, the hardware resource utilization is affected by query performance. Our proposed method can support concurrency by introducing the queuing theory in the memory latency and I/O latency model.

VII. Conclusions and Future Work

In this study, we proposed a cost calculation method for an in-memory DBMS using a disk-based DBMS. We focused on a CPU pipeline architecture and classified the CPU cycles into three types based on the operational characteristics of the front-end and back-end. The calculation formulas were created using data measured by the CPU-embedded performance monitor. In the evaluation of the two-table join, three-table join, and five-table join, the difference in selectivity corresponding to the intersection points of NLJ and HJ, between the proposed method and measurements, was increased in more than 90% from the conventional method. This means that the cost formulas can model the actual join operation with high accuracy. By applying the proposed cost calculation formulas, the proper join method can be selected and the risk of unexpected query execution delay for users of the DBMS can be reduced. In the future, we will evaluate different generation of CPUs and analyze how the differences in CPU architecture affect the cost formulas. We will also 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.

Appendix A
Queries for Evaluating Cost Calculation Formulas

The Queries used for evaluation of the proposed cost calculation formulas are shown in Figure A.1, A.2, A.3, and A.4.

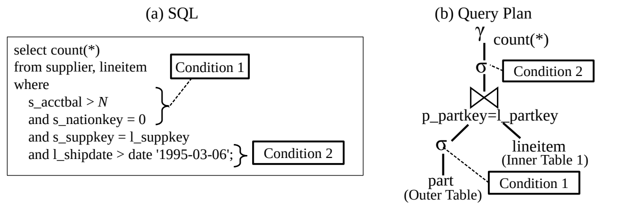


```
select count(*)
from  part, lineitem
where
  (p_type='STANDARD ANODIZED TIN'
  or p_type='STANDARD ANODIZED STEEL)
  and p_size < N
  and p_partkey = l_partkey
  and l_shipdate < date '1995-03-06';
```

(c) Selection Condition and Selectivity

N	6	8	10	20	30	40	50
Selectivity P_O (Condition 1)	1.33×10^{-3}	1.87×10^{-3}	2.40×10^{-3}	5.07×10^{-3}	7.73×10^{-3}	1.04×10^{-2}	1.31×10^{-2}

Fig. A.1. Target query of cost estimation for part and lineitem join

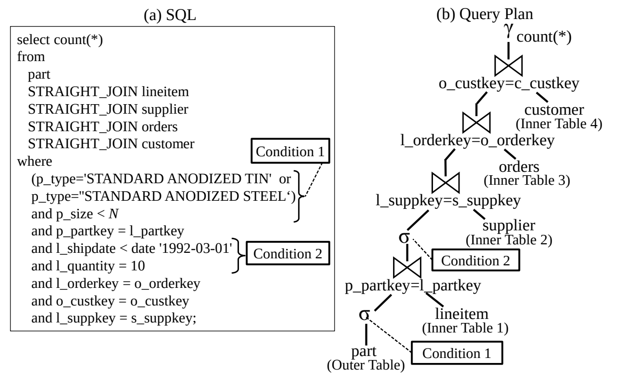


```
select count(*)
from supplier, lineitem
where
  s_acctbal > N
  and s_nationkey = 0
  and s_suppkey = l_suppkey
  and l_shipdate > date '1995-03-06';
```

(c) Selection Condition and Selectivity

N	9998	9978	9798	9200	9000	8000	7000
Selectivity P_O (Condition 1)	7.24×10^{-6}	8.00×10^{-5}	7.35×10^{-4}	2.91×10^{-3}	3.64×10^{-3}	7.27×10^{-3}	1.09×10^{-2}

Fig. A.2. Target query of cost estimation for supplier and lineitem join



```
select count(*)
from
  part
  STRAIGHT_JOIN lineitem
  STRAIGHT_JOIN supplier
  STRAIGHT_JOIN orders
  STRAIGHT_JOIN customer
where
  (p_type='STANDARD ANODIZED TIN'  or
  p_type="STANDARD ANODIZED STEEL‘)
  and p_size < N
  and p_partkey = l_partkey
  and l_shipdate < date '1992-03-01'
  and l_quantity = 10
  and l_orderkey = o_orderkey
  and o_custkey = o_custkey
  and l_suppkey = s_suppkey;
```

(c) Selection Condition and Selectivity

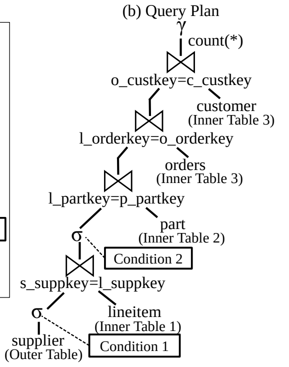
N	6	8	10	20	30	40	50
Selectivity P_O (Condition 1)	1.33×10^{-3}	1.87×10^{-3}	2.40×10^{-3}	5.07×10^{-3}	7.73×10^{-3}	1.04×10^{-2}	1.31×10^{-2}

Fig. A.3. Target query of cost estimation
for part, lineitem, supplier, orders, and customer join

(a) SQL

```
select count(*)
from
   supplier
   STRAIGHT_JOIN lineitem
   STRAIGHT_JOIN part
   STRAIGHT_JOIN orders
   STRAIGHT_JOIN customer
where
   s_acctbal > 9998            } Condition 1
   and s_nationkey = 0         }
   and s_suppkey = l_suppkey
   and l_shipdate > date '1995-03-06'} Condition 2
   and l_partkey = p_partkey
   and l_orderkey = o_orderkey
   and o_custkey = c_custkey;
```

(b) Query Plan



(c) Selection Condition and Selectivity

N	9998	9978	9798	9200	9000	8000	7000
Selectivity P_O (Condition 1)	7.24×10^{-6}	8.00×10^{-5}	7.35×10^{-4}	2.91×10^{-3}	3.64×10^{-3}	7.27×10^{-3}	1.09×10^{-2}

Fig. A.4. Target query of cost estimation
for supplier, lineitem, part, orders and customer join

Appendix B
Measured CPU Counters

The list of CPU counters for modeling the CPU cost calculation are shown in Table B.I. The constants and intermediate variable for modeling cost calculation formulas are shown in Table B.II. The column of "Symbol" means variables in the cost calculation formulas.

TABLE B.I. Lists of CPU Counters to Measure and Preprocess

No.	Counter Name
$E1$	CPU_CLK_UNHALTED.THREAD
$E2$	INST_RETIRED.ANY
$E3$	BR_MISP_EXEC.ANY
$E4$	DTLB_MISSES.ANY
$E5$	ITLB_MISS_RETIRED
$E6$	L1I.CYCLES_STALLED
$E7$	L1I.HITS
$E8$	L1I.MISSES
$E9$	L2_RQSTS.IFETCH_HIT
$E10$	L2_RQSTS.IFETCH_MISS
$E11$	MEM_INST_RETIRED.LOADS
$E12$	MEM_LOAD_RETIRED.HIT_LFB
$E13$	MEM_LOAD_RETIRED.L1D_HIT
$E14$	MEM_LOAD_RETIRED.L2_HIT
$E15$	MEM_LOAD_RETIRED.LLC_MISS
$E16$	MEM_LOAD_RETIRED.LLC_UNSHARED_HIT
$E17$	MEM_LOAD_RETIRED.OTHER_CORE_L2_HIT_HITM
$E18$	OFFCORE_RESPONSE.DATA_IFETCH.LOCAL_CACHE_1
$E19$	OFFCORE_RESPONSE.DATA_IFETCH.LOCAL_DRAM _AND_REMOTE_CACHE_HIT_0
$E20$	OFFCORE_RESPONSE.DATA_IFETCH.OTHER_LOCAL _DRAM_1
$E21$	OFFCORE_RESPONSE.DATA_IFETCH.REMOTE_CACHE _HITM_0
$E22$	OFFCORE_RESPONSE.DATA_IFETCH.REMOTE_DRAM_1
$E23$	OFFCORE_RESPONSE.DATA_IN.LOCAL_DRAM_AND _REMOTE_CACHE_HIT_0
$E24$	OFFCORE_RESPONSE.DATA_IN.OTHER_LOCAL_DRAM_0
$E25$	OFFCORE_RESPONSE.DATA_IN.REMOTE_CACHE_HITM_0
$E26$	OFFCORE_RESPONSE.DATA_IN.REMOTE_DRAM_1
$E27$	RESOURCE_STALLS.ANY
$E28$	RESOURCE_STALLS.LOAD
$E29$	RESOURCE_STALLS.ROB_FULL
$E30$	RESOURCE_STALLS.RS_FULL
$E31$	RESOURCE_STALLS.STORE
$E32$	UOPS_ISSUED.ANY
$E33$	UOPS_ISSUED.CORE_STALL_CYCLES
$E34$	UOPS_ISSUED.CYCLES_ALL_THREADS
$E35$	UOPS_ISSUED.FUSED
$E36$	UOPS_RETIRED.ANY

References

[1] T. Tanaka and H. Ishikawa, "Measurement-based cost estimation method of a join operation for an in-memory database," in *MMEDIA 2017, The Ninth International Conferences on Advances in Multimedia*. Venice, Italy: IARIA, April 2017, pp. 57–66.

[2] 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.

[3] 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].

[4] ——, "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].

[5] 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

[6] 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.

[7] O. Sandsta, "Mysql Cost Model," http://www.slideshare.net/olavsa/mysql-optimizer-cost-model [retrieved: March, 2017], October 2014.

[8] W. Du, R. Krishnamurthy, and M.-C. Shan, "Query optimization in a heterogeneous dbms," in *VLDB*, vol. 92, 1992, pp. 277–291.

[9] Q. Zhu and P.-A. Larson, "Building regression cost models for multidatabase systems," in *Proceedings of the Fourth International Conference on on Parallel and Distributed Information Systems*, ser. DIS '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 220–231. [Online]. Available: http://dl.acm.org/citation.cfm?id=382006.383210

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

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

[12] 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.

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

[14] 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.

[15] (2017) The MariaDB foundation - ensuring continuity and open collaboration in the mariadb ecosystem. [Online]. Available: https://mariadb.org/

[16] "TPC BENCHMARK™ 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.

[17] (2017) 7-Zip LZMA Benchmark. [Online]. Available: http://www.7-cpu.com/

[18] 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

[19] 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].

TABLE B.II. Lists of Constants and Intermediate Variable

No.	Symbol	Events	Value
$E37$	–	CPU frequency [GHz] (Xeon L5630)	2.13
$E38$	L_{L1}	L1I Latency [cycle]	4
$E39$	L_{L1}	L1D Latency [cycle]	4
$E40$	L_{L2}	L2 Latency [cycle]	10
$E41$	L_{LLLC}	Local LLC Latency [cycle]	40
$E42$	L_{RLLC}	Remote LLC Latency [cycle]	200
$E43$	L_{LMM}	Local Main Memory Latency [cycle]	$E41 + 67[ns] \times E37$
$E44$	—	Remote Main Memory Latency [cycle]	$E41 + 105[ns] \times E37$
$E45$	L_{MP}	Branchmiss prediction cycle	15
No.	Symbol	Events	Calculation Formula for Preprocessing
$E46$	I_{Load}	LOAD instruction	E11
$E47$	M_{L1D}	L1D Hit (data)	$E46 \times E13/(E12 + E13 + E14 + E15 + E16 + E17)$
$E48$	—	L1D Miss (data)	$E46 - E47$
$E49$	M_{L2D}	L2 Hit (data)	$E46 \times ((1-(E13/(E12+E13+E14+E15+E16+E17))) \times (E14/(E14+E15+E16+E17)))$
$E50$	—	L2 Miss (data)	$E48 - E49$
$E51$	M_{LLLCD}	LLC Hit (data)	$E46 \times ((1 - (E13/(E12 + E13 + E14 + E15 + E16 + E17))) \times (1 - (E14/(E14 + E15 + E16 + E17))) \times ((E16 + E17)/(E15 + E16 + E17)))$
$E52$	—	LLC Miss (data)	$E50 - E51$
$E53$	M_{RLLCD}	Remote LLC Hit (data)	$E46 \times ((1-(E13/(E12+E13+E14+E15+E16+E17))) \times (1-(E14/(E14+E15+E16+E17))) \times (1-((E16+E17)/(E16+E17+E15))) \times ((E23+E25)/(E23+E24+E25+E26)))$
$E54$	M_{LMMD}	Local Main Memory (data)	$E46 \times ((1 - (E13/(E12 + E13 + E14 + E15 + E16 + E17))) \times (1 - (E14/(E14 + E15 + E16+E17))) \times (1-((E16+E17)/(E16+E17+E15))) \times (E24/(E23+E24+E25+E26)))$
$E55$	—	Remote Main Memory (data)	$E46 \times ((1 - (E13/(E12 + E13 + E14 + E15 + E16 + E17))) \times (1 - (E14/(E14 + E15 + E16+E17))) \times (1-((E16+E17)/(E16+E17+E15))) \times (E26/(E23+E24+E25+E26)))$
$E56$	—	Total Data Access Latency	$E47 \times E39 + E49 \times E40 + E51 \times E41 + E54 \times E43 + E55 \times E44 + E53 \times E42$
$E57$	I	Instruction	$E2 + E3$
$E58$	M_{L1I}	L1I Hit (instruction)	$E57 - E8$
$E59$	—	L1I Miss (instruction)	$E57 - E58$
$E60$	M_{L2I}	L2 Hit (instruction)	$E8 - E10$
$E61$	—	L2 Miiss (instruction)	$E59 - E60$
$E62$	M_{LLLCI}	Local LLC Hit (instruction)	$E10 \times ((E18/(E18 + E19 + E21 + E20 + E22)))$
$E63$	—	Local LLC Miss (instruction)	$E61 - E62$
$E64$	M_{RLLCD}	Remote LLC Hit (instruction)	$E57 \times ((E10/E57) \times (((E19 + E21)/(E18 + E19 + E20 + E21 + E22))))$
$E65$	M_{LMMD}	Local Main Memory (instruction)	$E10 \times ((E20/(E18 + E19 + E20 + E21 + E22))))$
$E66$	—	Remote Main Memory (instruction)	$(E33 - E27) \times ((E10/E57) \times (E22/(E18 + E19 + E20 + E21 + E22)))$
$E67$	—	Total Instruction Access Latency	$E60 \times E40 + E62 \times E41 + E64 \times E42 + E65 \times E43 + E66 \times E44$
$E68$	$C_{DCacheAcc}$	Data Access	$E27 + E34$
$E69$	C_{MP}	Branch Mispredition Penalty	$E3 \times E45$
$E70$	$C_{ICacheMiss}$	Instruction Penalty	$E33 - E27 - E69$

[20] 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.

[21] A. Rahal, Q. Zhu, and P.-A. Larson, “Evolutionary techniques for updating query cost models in a dynamic multidatabase environment,” *The VLDB Journal*, vol. 13, no. 2, pp. 162–176, May 2004. [Online]. Available: http://dx.doi.org/10.1007/s00778-003-0110-4

[22] Q. Zhu, S. Motheramgari, and Y. Sun, “Cost estimation for large queries via fractional analysis and probabilistic approach in dynamic multidatabase environments,” in *Proceedings of the 11th International Conference on Database and Expert Systems Applications*, ser. DEXA ’00. London, UK, UK: Springer-Verlag, 2000, pp. 509–525. [Online]. Available: http://dl.acm.org/citation.cfm?id=648313.755672

[23] 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, pp. 231–246, 2000.

[24] 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.

[25] M. S. Kester, M. Athanassoulis, and S. Idreos, “Access path selection in main-memory optimized data systems: Should I scan or should I probe?” in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD ’17. New York, NY, USA: ACM, 2017, pp. 715–730. [Online]. Available: http://doi.acm.org/10.1145/3035918.3064049