Improvement of Cost Calculation Accuracy for Database Query Optimization by Introducing Performance Model Based on CPU Architecture

CPU アーキテクチャを考慮した性能モデルの導入によるデータベース・クエリ最適化のためのコスト計算の精度向上

2019 年 3 月

田 中　剛

首 都 大 学 東 京

Abstract

Non-volatile memory is applied not only to storage subsystems but also to the main memory of computers to improve performance and increase capacity. In the near future, some in-memory database systems will use non-volatile main memory as a durable medium instead of using existing storage devices, such as hard disk drives or solid-state drives. In addition, cloud computing is gaining more attention, and users are increasingly demanding performance improvement. In particular, the Database-as-a-Service (DBaaS) market is rapidly expanding. Attempts to improve database performance have led to the development of in-memory databases using non-volatile memory as a durable database medium rather than existing storage devices.

For such in-memory database systems, the cost of memory access instead of Input/Output (I/O) processing decreases, and the Central Processing Unit (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 most appropriate join method, the query execution time increases. Moreover, in the cloud computing environment the CPU architecture of different physical servers may be of different generations. The cost model is also required to be capable of application to different generation CPUs through minor modification in order not to increase database administrator's extra duties.

To improve the accuracy of the cost calculation, a cost calculation method based on CPU architecture using statistical information measured by a performance monitor embedded within the CPU (hereinafter called measurement-based cost calculation method) is proposed, and the accuracy of estimating the intersection (hereinafter called cross point) of cost calculation formulas for join methods is evaluated.

In this calculation method, we concentrate on the instruction issuing part in the instruction pipeline, inside the CPU architecture. The cost of database search processing is classified into three types, data cache access, instruction cache miss penalty and branch misprediction penalty, and for each a cost calculation formula is constructed. Moreover, each cost calculation formula models the tendency between the statistical information measured by the performance monitor embedded within the CPU and the selectivity of the table while executing join operations. The statistical information measured by the performance monitor is information such as the number of executed instructions and the number of cache hits. In addition, for each element separated into elements repeatedly appearing in the access path of the join, cost calculation formulas are formed into parts, and the cost is calculated combining the parts for an arbitrary number of join tables.

First, to investigate the feasibility of the proposed method, a cost formula for a two-table join was constructed using a large database, 100 GB of the TPC Benchmark™H database. The accuracy of the cost calculation was evaluated by comparing the measured cross point with the estimated cross point. The results indicated that the difference between the predicted cross point and the measured cross point was less than 0.1% selectivity and was reduced by 71% to 94% compared with the difference between the cross point obtained by the conventional method and the measured cross point. Therefore, the proposed cost calculation method can improve the accuracy of join cost calculation.

Then, to reduce the operating time of the database administration, the cost calculation formula was constructed under the condition that the database for measuring the statistical value was reduced to a small scale (5 GB). The accuracy of cost calculations was also evaluated when joining three or more tables. As a result, the difference between the predicted cross point and the measured cross point was reduced by 74% to 95% compared with the difference between the cross point obtained by the conventional method and the measured cross point. It means the proposed method can improve the accuracy of cost calculation.

Finally, a method is also proposed for updating the cost calculation formula using the measurement-based cost calculation method to support a CPU with architecture from another generation without requiring re-measurement of the statistical information of that CPU. Our approach focuses on reflecting architectural changes, such as cache size and associativity, memory latency, and branch misprediction penalty, in the components of the cost calculation formulas. The updated cost calculation formulas estimated the cost of joining different generation-based CPUs accurately in 66% of the test cases.

In conclusion, the in-memory database system using the proposed cost calculation method can select the best join method and can be applied to a database system with CPUs from different generations.

Contents

1. Introduction **1**

2. Related Works **6**

2.1. Estimating Data Distribution for Query Optimization 6
2.2. Cost Calculation 7
2.2.1. Single Instance 7
2.2.2. Multiple Instance 10
2.3. Hardware Activity Evaluation on Database Workload 10
2.4. Hardware Modeling 11

3. Proposed CPU Cost Model **12**

4. DBMS Operation Model **18**

5. Cost Calculation Formula **24**

6. Feasibility Study of Creating Cost Calculation Formulas **31**

6.1. CPU Events Measurement for Parameter Setting of Cost Calculation Formulas 31
6.2. Measurement Results and Cost Calculation Formulas 33
6.2.1. Measurement Results of NLJ 33
6.2.2. Measurement Results of HJ Build Phase 41
6.2.3. Measurement Results of HJ Probe Phase 43
6.3. Evaluation of Cost Calculation Results 48
6.4. Discussion 50
6.5. Conclusion of Feasibility Study 52

7. Measurement-based Cost Estimation Method for Multi-Table Join Operation **53**

7.1. Proposal and Verification of Measurement-based Cost Estimation Method . . 53
7.1.1. Measurement of Parameters for Creation of Cost Formula 53
7.1.2. Measurement Results and Cost Calculation Formulas of Join Operation 55
7.2. Degradation of Data Distribution Accuracy 76
7.3. Verification of Determining Join Method and Ordering Joining Tables 77
7.3.1. Verification Method 77
7.3.2. Comparison of Measured Query Execution Time and Estimated Query Cost 79
7.4. Discussion 83
7.5. Conclusions 85

8. Updating Cost Calculation Method Applying to Different Generation CPUs 86
8.1. Cost Calculation Method for Join Operation 87
8.2. Extension of Cost Calculation Method 89
8.3. Evaluation Method 92
8.4. Results and Discussion 93
8.5. Conclusion 96

9. Conclusions and Future Works 99

A. Appendix 108
A.1. Queries for Evaluating Cost Calculation Formulas 108
A.2. Measured CPU Counters 110

List of Figures

1.1. Disk-based Database and In-memory Database 2
1.2. Cost Estimation Problem for the Selection of Join Methods 3
1.3. Cloud Service Infrastructure 4

2.1. Relation of Data Access Pattern and Cache Hit 9

3.1. Focus point of the CPU pipeline 12

4.1. Degradation and Split Cost Calculation Method 19
4.2. NLJ after HJ Case 19
4.3. Split Cost Calculation of Grouping, Aggregating, Sort, etc. 20
4.4. Sort Merge Join Case 21
4.5. Calculating Method of Statistics information of Sorting Data Stored in Temporary Table for Sort-Merge Join 21
4.6. Dividing TPC-H Query3 into Cost Calculation Parts 23

5.1. Input Parameters to Each Part of Sort Merge Join 28

6.1. Target Query of Measurement and Cost Estimation 32
6.2. Number of Instructions on NLJ and Regression Line 34
6.3. Number of Instructions and L1I Hit Ratio on NLJ 35
6.4. Estimation Method of Curves Using Dummy Variable 35
6.5. Instruction Access Ratio of L2, LLC and Main Memory on NLJ 37
6.6. Data Access Ratio of L2, LLC and Main Memory on NLJ 38
6.7. Relation between Number of Inner Table Records and L1D Hit Ratio on NLJ 39
6.8. Relation between Sum of Memory Accesses and Instruction Miss Penalty Cost or Data Cache Access Cost on NLJ 40
6.9. Relation between Selectivity and Branch Misprediction Cycle on NLJ 40
6.10. Slope and Intercept of Regression Line of Data Access Cycles and Branch Misprediction on NLJ 41
6.11. Number of Instruction References and Data References in Build Phase of HJ . 43
6.12. Slope and Intercept of Regression Line of Data Access Cycles and Branch Misprediction in Build Phase of HJ 44
6.13. Number of Instruction References and Data References in Probe Phase of HJ . 45
6.14. Slope and Intercept of Regression Line of Data Access Cycles and Branch Misprediction in Probe Phase of HJ 46
6.15. Comparison of Measured Results, Proposed Cost Model and Conventional Cost Model 49

6.16. Comparison of Measured Results, Proposed Cost Model and Conventional Cost Model Considering CPU Cycle Time or Main Memory Latency 51

7.1. Target Query of Measurement and Cost Estimation for Two-table Join 56
7.2. Target Query of Measurement and Cost Estimation for Three-table Join 57
7.3. CPU Event Count on Executing NLJ 59
7.4. CPU Event Count on Executing Build Phase of HJ 60
7.5. CPU Event Count on Executing Probe Phase of HJ 61
7.6. CPU Event Count on Combination Probe Phase 62
7.7. Problem of Modeling Cache Hit Ratio 63
7.8. Simple Example of Number of Records Having the Same Key Value (rsk) 64
7.9. Number of Disk I/Os and Disk I/O Processing Time Ratio while Joining Customer Table and Order Table by NLJ and HJ 67
7.10. Difference of Number of Disk I/Os in Joining Table Size by NLJ 68
7.11. Number of Synchronous Disk I/O Count during Combination Probe Phase . . 68
7.12. Cost Comparison of Measured, Proposed Cost Model, and Conventional Cost Model Results for Joining Two Tables 71
7.13. Cost Comparison of Measured, Proposed Cost Model, and Conventional Cost Model Results for Joining Three or More Tables 72
7.14. Cost Comparison of Measured, Proposed Cost Method and Updated Conventional Cost Method Results for Joining Two Tables 74
7.15. Cost Comparison of Measured, Proposed Cost Method, and Updated Conventional Cost Method Results for Joining Three or More Tables 75
7.16. Histograms Using Difference of Measured Cross Point and Proposed Cross Point 76
7.17. Queries for Evaluating Join Method and the Order of Joining Tables 78
7.18. Distribution of Number of I/O Requests in Nested Loop Join 81
7.19. Ratio of Size of Issued I/O requests in Nested Loop Join 82
7.20. I/O Request Size in Joining Orders and Lineitem by Nested Loop Join 83
7.21. NLJ after HJ Case 84

8.1. Overview of Query Execution Flow Using Measurement-based Cost Calculation 86
8.2. Difference in Updating Architectural Changes for the Skylake Cost Model Based on Measurement Values of Westmere 94
8.3. Difference in Updating Architectural Changes for the Westmere Cost Model Based on Measurement Values of Skylake 96
8.4. Decomposition of the Join Costs of the C–O and C–O–L Tables 97
8.5. Reason for Lower Accuracy of Estimated Cross Points after Model Modification 98

A.1. Target Query of Cost Estimation for Part and Lineitem Join 108
A.2. Target Query of Cost Estimation for Supplier and Lineitem Join 109
A.3. Target Query of Cost Estimation for Part, Lineitem, Supplier, Orders, and Customer join 109

A.4. Target Query of Cost Estimation for Supplier, Lineitem, Part, Orders and Customer join 110

List of Tables

2.1. Explanatory Variables for Creating Cost Calculation Formulas 10

3.1. Notations for CPU Cost Calculation Model (1) 14
3.2. Notations for CPU Cost Calculation Model (2) 15
3.3. Notations for CPU Cost Calculation Model (3) 16

5.1. Parameter List for Cost Calculation 24
5.2. Parts of SMJ and Notation of Cost 28
5.3. Parameter Estimation Methods 30

6.1. Measurement Environment for Feasibility Study 32
6.2. Coefficient of Determination of Regression Line of Instruction Access on NLJ 36
6.3. Parameter Setting of NLJ 42
6.4. Parameter Setting of HJ 47
6.5. Difference of Cross Point and Improvement Ratio 48
6.6. Difference of Cross Point and Improvement Ratio with Updated Conventional Method 50

7.1. Parameter Measurement Environment 54
7.2. Slope and Intercept of the Regression Models 69
7.3. Slope and Intercept of the Regression Models in Combination Probe Phase . . 70
7.4. Accuracy Improvement Ratio for Estimating Cross Point of NLJ and HJ . . . 73
7.5. Accuracy Improvement Ratio for Estimating Cross Point of NLJ and HJ with Updated Conventional Method 73
7.6. Increasing Ratio of Equi-Width Histogram buckets Width Using Updated Conventional Method and Proposed Method 77
7.7. Evaluation Environment 79
7.8. Measured Time and Estimated Cost to Decide a Join Method and Order of Accessing Tables 80

8.1. Architectural Difference between CPU Generations 89
8.2. Evaluation Environment 93
8.3. Accuracy of Estimating the Selectivity of the Cross Point of a Nested Loop Join and Hash Join when Applying the Westmere-based Model to Skylake . . 94
8.4. Accuracy of Estimating the Selectivity of the Cross Point of a Nested Loop Join and Hash Join when Applying the Skylake-based Model to Westmere . . 95

A.1. Lists of CPU Counters to Measure and Preprocess of Westmere Processor . . 111

A.2. Lists of Constants of Westmere Processor 112
A.3. Lists of Intermediate Variable of Westmere Processor 113
A.4. Lists of CPU Counters to Measure and Preprocess of Skylake Processor . . . 114
A.5. Lists of Constants of DB Server with Skylake Processor 114
A.6. Lists of Intermediate Variable of Skylake Processor 115

1. Introduction

Improving the performance and expanding the capacity of non-volatile memory (NVM) is necessary for both high-speed disk drives and main memory units. Accordingly, Intel and Micron developed an NVM called 3D Xpoint memory [1]. An NVM is implemented as a byte-addressable memory and assigned as a part of the main memory space. An application programming interface (API) [2] for accessing the NVM was proposed to make the development of applications easier. 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.

Tera-byte class in-memory databases are proposed to provide expanded capacity and performance improvement, and for the emergence of Dual Inline Memory Module (DIMM) compatible implementation of NVM [3, 4]. An in-memory database is intended for use in decision support systems. Therefore, our main target operations are analytical queries such as those of the TPC-H benchmark [5].

There are two types of implementations of in-memory databases through the application of an 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 1.1(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 1.1(a)) are moved to files on the NVM using libraries and accessed by API for the NVM [6] (Figure 1.1(b)). When accessing the database, the performance of the load/store type is better than that of 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) need not be changed, which indicates that it is easy for the administrators to introduce an in-memory database system.

DBMS's query optimization methods include cost-based and rule-based. The cost based-optimization is a method of obtaining the minimum cost access path using the value of cost calculated from the distribution of data and data selection condition. The cost is a numerical value corresponding to the execution time of the query. The rule based-optimization is the method of obtaining the access path according to a rewrite rule of the description of the query. It is possible to provide a stable access path irrespective of changes in the distribution characteristics of data. However, to keep up with fluctuation of data distribution, database administrator tuning work is required. In a large-scale system such as cloud services, a cost-based optimization is better to reduce the operation workload because it is possible to optimize queries automatically. Therefore, this study target is the cost-based optimization.

The DBMS encounters a problem when preparing for the execution of an analytic query.

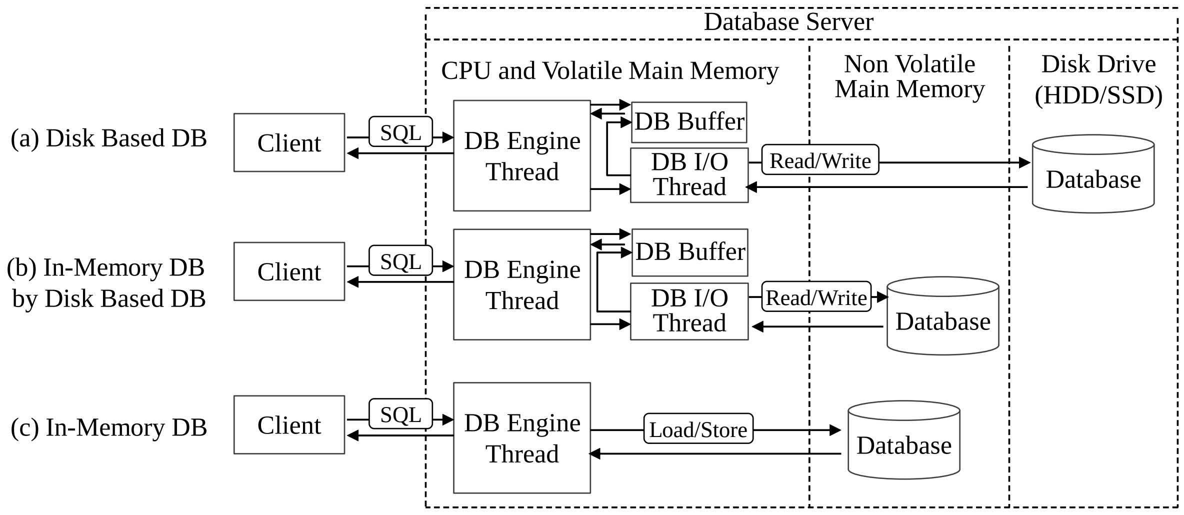


Figure 1.1.: Disk-based Database and In-memory Database

In general, the DBMS performs several steps prior to query execution. First, it analyzes the query. Second, it creates multiple access paths. Third, it estimates the query processing cost for each access path. Finally, it selects the access path with the minimum cost from several candidates. For example, when the DBMS joins two tables, such as the R table and S table shown in Figure 1.2(a), it generates the access path (Figure 1.2(b)) that minimizes the number of rows to be referenced. At this time, the execution time depends on the join method selected by the DBMS. 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 joining 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 1.2, the selectivity is determined by the condition x for column R.C in Figure 1.2(c). In Figure 1.2(c), two cost functions intersect at X_{cross}. Join method 2 must be chosen from the left side of X_{cross}, and join method 1 must be chosen from the right side of X_{cross}. If the DBMS cannot estimate the selectivity X_{cross} accurately, it will choose the wrong join method.

In the cost-based optimization, both the accuracy of estimating the data distribution and the accuracy of the cost calculation formula has been required to improve the cost accuracy. Many studies have been done on improving the accuracy of data distribution such as histograms [7]. However, it is also required to increase the accuracy of the cost calculation formula together [8]. In this research we will examine improvement of accuracy.

In general, the query execution cost is expressed as the sum of the central processing unit (CPU) cost and the I/O cost [9, 10]. The CPU cost is the CPU time, whereas the I/O cost is the latency when accessing the disk drive. 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 1.1(c)), it becomes necessary to predict the CPU cost accurately.

To improve the accuracy of prediction of CPU processing cost, the estimation of CPU

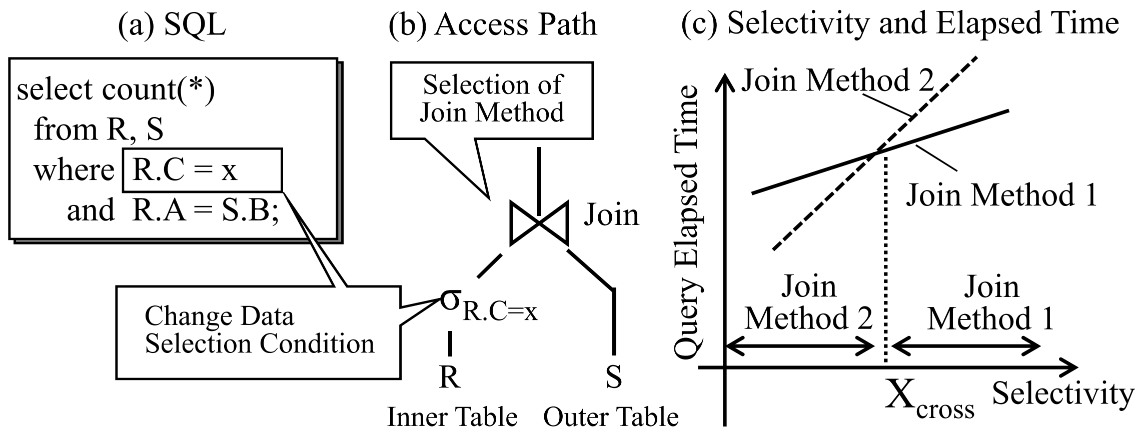


Figure 1.2.: Cost Estimation Problem for the Selection of Join Methods

processing time must become more accurate than that obtained with the aforementioned conventional method. 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. To improve the accuracy of cost calculation, we focus on constructing a CPU operation model by considering the CPU architecture.

In this study, we propose a method based on statistical information on CPU operations to improve the accuracy of estimation of CPU cost for in-memory databases applied to existing DBMSs (Figure 1.1(b)) and native in-memory databases (Figure 1.1(c)).

As mentioned above, our main target operations are analytic queries. An analytic query is composed of selection, projection, join, aggregation, sort, etc. The query optimizer decides the order of accessing tables and chooses a proper join method and does not need to choose an alternative way for other operations. For instance, a quick sort algorithm is commonly used in most DBMSs. The optimizer does not need to choose different sort algorithms such as merge sort and bubble sort. This study focuses on choosing a proper join method with selection operation. The proper join method depends on the selectivity of the attributes in `WHERE` clause in the query . The order of tables is determined according to the traditional way based on the number of records to be accessed [9].

Processing time increases in proportion to the number of accessed records. However, the number of accessed records and the record access patterns, such as random or sequential, are different for each join method. The analytic query processing time increases depending on which method is selected. Therefore, the cost calculation formula of join processing is our main research target. Our proposed cost calculation method can be easily applied to operations other than join if the CPU statistical information can be measured.

On the other hand, the market for cloud computing is expanding. In particular, the demand for database services in the cloud (DataBase-as-a-Service; DBaaS) is rapidly increasing. DBaaS requires higher performance and reliability than other cloud-based services. Its performance requirements can be satisfied by using NVM as a durable database medium instead of existing storage devices. In-memory databases using NVM as the main memory are expected to gain popularity. When the user requests a new virtual server from the cloud service provider, an image of a virtual machine (VM) as an in-memory database server is attached to the physical server, as shown in Figure 1.3.

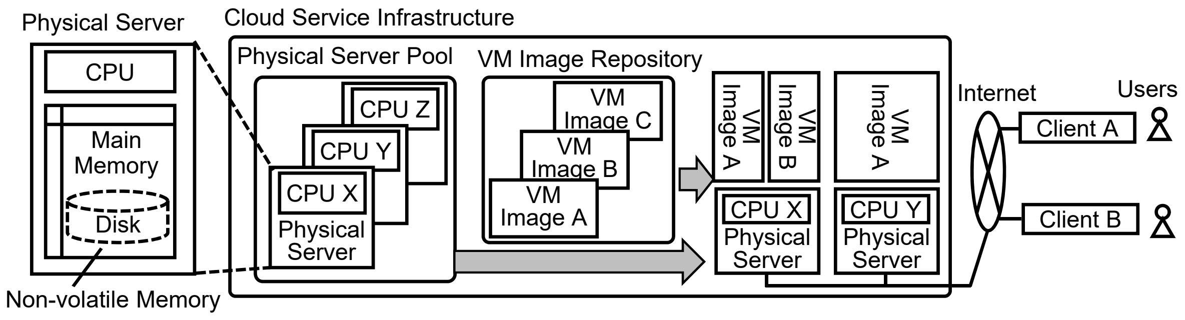


Figure 1.3.: Cloud Service Infrastructure

In the cloud environment, many DBMS instances run concurrently. We solve the single instance case in this study. We will solve the multi instance case in the future works. The cost calculation equations of multi instance case can be created by modifying those of single instance case.

From the viewpoint of the operation cost of the cloud environment, it is unrealistic to recreate the cost calculation formula every time the VM with the database is transferred to another server using CPUs with a different architecture. This study aims to propose methods to obtain cost calculation formulas that can employ CPUs with different architectures with no changes or minor changes. The change in performance as a result of architectural changes, such as the memory latency and cache size, among different generations of CPUs is reflected in the proposed cost calculation formula. We used this updated cost calculation formula to verify whether it is possible to select the joining methods, i.e., nested loop join (NLJ) and hash join (HJ), accurately for different generations of CPUs. We determined that the updated cost calculation formulas can estimate the cross point accurately.

In this study, our proposed highly accurate cost calculation method allows portability of the cost calculation formulas across different generations of CPUs and can contribute to reducing the cost of cloud service platforms.

The rest of this paper is organized as follows:

- In chapter 3, 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 in chapter 4. Chapter 5 describes inputs and outputs of the cost calculation method and shows the overall view of the cost calculation formulas for join operations such as NLJ and HJ.

- Chapter 6 describes a feasibility study of our proposed cost calculation method. Our first target is to propose cost calculation methods of two-table join using large data. 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. 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.
- In chapter 7, in developing the cost calculation formula, reference queries are executed on a small reference database. The cost calculation formulas for multiple-table join are proposed, and the accuracy of estimating the cross point of NLJ and HJ is evaluated. Moreover, we verify that the cost calculation formulas can be used to determine the order of tables to be joined.
- In chapter 8, we propose a method for extending the join operation cost calculation formulas for different generations of CPUs. By considering the differences in CPU pipeline architectures between CPUs of different generations, we classify the architectural changes and impacts on performance. Using this classification, we propose an extending method of the measurement-based cost calculation formula. We measure the statistic information for executing the join operation and obtain cost calculation formulas of the CPUs of different generations using the measurement results. We verify the accuracy of estimating the cross point using the cost calculation formula of the target CPU obtained from a reference CPU.
- Finally, in chapter 9, we summarize the conclusions and describe future works.

2. Related Works

There are two approaches to optimize queries of database. One is cost-based optimization, another is rule-based optimization. The rule-based optimization is a method in which the access path is statically determined based on the rule and it is often used in banking systems that is required to avoid sudden system behavior change by the access path of queries change. On the other hand, the cost-based optimization has the advantage that it can automatically optimize following the characteristics of data if the distribution of data changes.

This study relates to cost calculation methods used for cost-based query optimization of database. In the optimization of the query, it is necessary to accurately obtain the selectivity which is the input of the cost calculation. In order to correctly obtain the selectivity, it is necessary to understand the related studies for managing the statistical information of the data distribution represented by the histogram in the database. Therefore, this chapter introduces the related studies on the statistical information of the data distribution stored in the database and the related studies on the cost calculation. In addition, we will introduce the related works on the modeling of CPU behavior which is another problem of this study.

2.1. Estimating Data Distribution for Query Optimization

The selectivity used for cost calculation can be estimated from the frequency distribution of attributes of the database. Several techniques have been proposed to estimate the frequency distribution [11]. Many commercial DBMSs use histogram [12].

A histogram is one of means for expressing distribution of data and is created by dividing the data distribution of attributes into β mutually disjoint subsets called buckets. Each bucket has approximating frequencies and values obtained using methods of extracting characteristics of data distribution [7].

Numerous types of histograms and their various construction methods have been proposed. Poosala *et al.* [13] introduces three viewpoints $p(s,u)$ called partition constraint (p), sort parameter (s) and source parameter (u) to classify various histograms. They defined spreads (S), attribute values (V), frequencies (F), cumulative frequencies (C), and area (A) as the sort parameters and source parameters. Spreads S is the distance between attribute values. Area A is given by the product of spreads S and frequencies F. In the following, some example of the histograms are introduced.

Equi-width [13] When dividing the data distribution into β buckets, the attribute value width of the data distribution is divided into equal width by $\beta-1$. Many commercial DBMSs use *equi-width*. The partition constraint of this histogram is $p(V,S)$.

Equi-depth [14] *Equi-depth* histogram is the sum of the frequencies in each bucket to be equal height. MySQL [15], MariaDB [16] and Oracle [7] use *equi-depth*. The partition constraint is $p(V, F)$.

V-Optimal [13] *V-Optimal* histogram is contiguous sets of frequencies into buckets so as to minimize the variance of the overall frequency approximation. The partition constraint is $p(F, F)$, $p(V, F)$, $p(V, A)$, $p(A, A)$ and $p(V, C)$.

V-Optimal-End-Biased [14] *V-Optimal-End-Biased* histogram is some of the highest frequencies and some of the lowest frequencies are placed in individual buckets, while the remaining frequencies are all grouped in a single bucket. The partition constraint is $p(F, F)$.

Maxdiff [13] *Maxdiff* histogram has a bucket boundary between two source parameter values that are adjacent if the difference between these values is one of the $\beta - 1$ largest such differences. The partition constraint is $p(V, F)$, $p(V, A)$ and $p(A, A)$. The *maxdiff* histogram with $p(V, A)$ is the best histogram on the issues of construction time and generated error.

Moreover, Poosala *et al.* proposed the multi-dimensional *maxdiff* histograms computed using the MHIST algorithm for accurate estimation of multi-dimensional data distribution, that is, combination of attributes [17]. To shorten time of creating histogram, sampling is used instead of searching whole data. Chaudhuri *et al.* proposed the calculation method of the number of samples to create *equi-depth* histogram [18]. On the other hand, Ioannidis *et al.* reported that the error of estimating size of query results increases exponentially with the number of joins [19]. Leis *et al.* determined that the contribution of selectivity is limited to improve query performance [8]. These researches suggests that not only effort of improvement accuracy of cost calculation method but also improvement accuracy of estimating distribution of attributes of data are required. We focused on improving accuracy of the cost calculation as the first step. We will tackle develop a method of estimating data distribution suitable for the proposed cost calculation method as a future work.

2.2. Cost Calculation

There has been many studies regarding how to calculate cost of executing a query. Cost calculation formulas include one assuming a state where a single DBMS instance is running and one assuming an environment where multiple instances are running. In addition, the single instance cases are further classified as white-box analysis [20] and black-box analysis [21].

2.2.1. Single Instance

White-box Analytic Approach

The white-box analytic approach is a method for creating cost calculation formulas by modeling the data access of the DBMS while executing the query. Based on this white-box analytic approach, there are some cost calculation methods. One is the product of a unit cost and the number of accessed records [9, 10, 22]. Some cost calculation model are the sum of only I/O cost [22] and others are the sum of CPU cost and I/O cost [9, 10].

The cost calculation formula is the sum of CPU cost and I/O cost as following:

$$cost = cpu\_cost + io\_cost. \tag{2.1}$$

For example, the cost formula for MySQL is given below [23]. The cost of scanning a table R is given by

$$table\_scan\_cost(\mathrm{R}) = record(\mathrm{R}) \times CPR + page(\mathrm{R}) \times CPIO \tag{2.2}$$

where *record*(R) is the number of records of table R, *CPR* is the CPU cost per record, *page*(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

$$\begin{aligned} table\_join\_cost(\mathrm{R}, \mathrm{S}) = table\_scan\_cost(\mathrm{R}) + record(R) \\ \times selectivity \times records\_per\_key(\mathrm{S}) \times (CPIO + CPR) \end{aligned} \tag{2.3}$$

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*(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.

Moreover, another method is to improve accuracy using the unit cost estimated from the execution time of several evaluation queries [10, 20].

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.

Another study on the micro-level approach is the method that applies a CPI measurement and focuses on a memory reference for cost calculations (2.4) of an in-memory database [24, 25]. In equation 2.4, *blocking factor* means the ratio of overlapped memory accesses. This research targeted a DBMS that use the load/store type memory access (Figure 1.1(c)). In this work, the number of cache hits or main memory accesses was predicted from the data access pattern of the database (Figure 2.1), 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 past studies, 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.

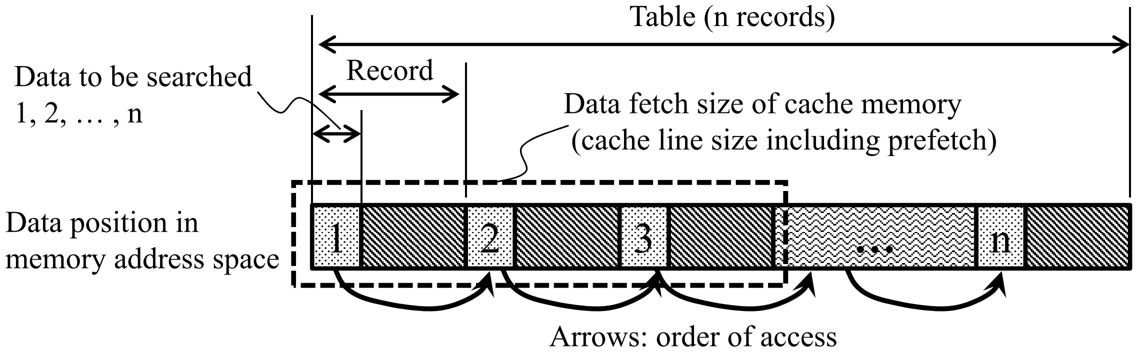


Figure 2.1.: Relation of Data Access Pattern and Cache Hit [25]

$$CPI = CPI0 + \{ \sum_{L2cache}^{last\ level\ cache} ((number\ of\ cache\ hits) \times latency \times (blocking\ factor)) + (number\ of\ main\ memory\ accesses) \times latency \times (blocking\ factor)\} \quad (2.4)$$

In many existing studies [9], the performance of queries was considered as a function of selectivity. Kester *et al.* [26] 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. In this study, the single query executing case is solved as first step. The cost calculation equations of multi query executing concurrently case will be able to be created by modifying those of single instance case as a future work.

Black-box Analytic Approach

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 as shown in Table 2.1 [21].

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 [21].

The cost value is relatively small at the cross point between NLJ and HJ. When the number of records in the outer table is RO and the number of records in the internal table is RI and the selectivity of the outer table P_O, then the number of records accessed by NLJ is $RO \times P_O \times RI$, that of HJ is $RO+RI$. In general, the cost value is proportion to the number of records to be accessed, the P_O is very small,[1] that is, the cost value is very small.

[1]Note. Let $RO = RI = 10^4$. If (NLJ execution time) < (HJ execution time), then $P_O < 2\times10^{-4}$.

Table 2.1.: Explanatory Variables for Creating Cost Calculation Formulas [21]

No.	Explanatory Variables
1	Cardinality of table of processing result of query
2	Size of intermediate result
3	Record length of table to be processed
4	Record length of obtained records by query
5	Number of used disk blocks of table to be processed and obtained records by query
6	System parameters such as number of process and memory size
7	Characteristics of an index such as height of number of leaves

Therefore, this study aims to calculate an accurate cost when the cost value is small, by modeling the CPU activities.

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 [27, 28]. Applying these methods to our approach will help achieve a more accurate model.

2.2.2. Multiple Instance

Our cost calculation model is based on the statistic information of CPU under a single VM execution. However, multiple VMs are executed on a real production system. Kester *et al.* [26] make models by multivariate regression of measured logical I/O latency when the plural of queries execute concurrently. 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, 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. Moreover, the concurrent query execution model is utilized for deciding the combination of executing queries parallelly to make batch operation time minimum [29].

2.3. Hardware Activity Evaluation on Database Workload

Ailamaki *et al.* [30] studied that 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 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. Hankins *et al.* [31] also studied characteristics of database workload such as TPC-C for online transaction processing using CPI. The largest component of CPI is main memory access occurred by L3 cache miss. TLB miss is less than half of branch misprediction penalty.

There is research that applied a CPI calculation method focusing on a memory reference to cost calculation (2.4) for an in-memory database [24] [25]. This research targets DBMS that use the load/store type memory access (Figure 1.1(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. As mentioned in Section 2.2, the way of obtaining *CPI0* is a difficult problem. Another CPI calculation method considering memory latency and the number of memory accesses of CPI is required.

From the point of cloud computing environment, Tanaka *et al.* [32] researched the database application whose performance bottleneck is disk I/O and CPU utilization is low, e.g. TPC-H, is suitable for virtualized environment. However, their research target is only analysis of performance evaluation using CPI and it has not predicted query performance.

2.4. Hardware Modeling

CPI is one of the most popular performance metric to evaluate CPU performance bottleneck.

As CPI-based evaluation method, the performance monitor embedded in the CPU is utilized while executing an application such as a benchmark program and measure statistical information such as the number of CPU execution cycles and the number of executed instructions and calculate CPI using these statistical information [30, 33, 34]. As other approach, simulating the CPU operation which execute a lightweight benchmark program or instruction sequence extracted only for the main part of the application program acquired beforehand by a method such as tracing [35, 36] and the desk study using a spreadsheet [37, 38], which are made to operate on the simulator based on the information obtained from the simulator.

These methods have advantages and disadvantages. Measurement with the performance monitor has the advantage of running an actual application, but it has restrictions to measure with the number of pieces of statistical information to be collected and the number and size of counters installed in the circuit [39]. The evaluation using simulators has the advantage that it can change the configuration of the hardware such as the size and the latency of the cache memory easily [35], but it is difficult to strictly evaluate a relatively large application such as a database. The desk study approach often use queuing theory, although it is relatively easy to evaluate to change the configuration of hardware easily as simulation, it is difficult to consider transient phenomena happened on operating system, device drivers and CPU.

In addition, when focusing on the cache miss penalty as in the Equation 3.2, it is popular to evaluate the memory latency constituting the CPI as a fixed value [35,40,41]. Memory latency become longer when the utilization of system resources such as main memory and disk drives become higher.

In the case of using CPI for performance evaluation of virtual environment [32,42], performance prediction using evaluation results has not been achieved. On the other hand, CPI is used for not only performance evaluation but also operation optimization of the VM that runs on the cloud environment [43].

3. Proposed CPU Cost Model

In this chapter, 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.

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.

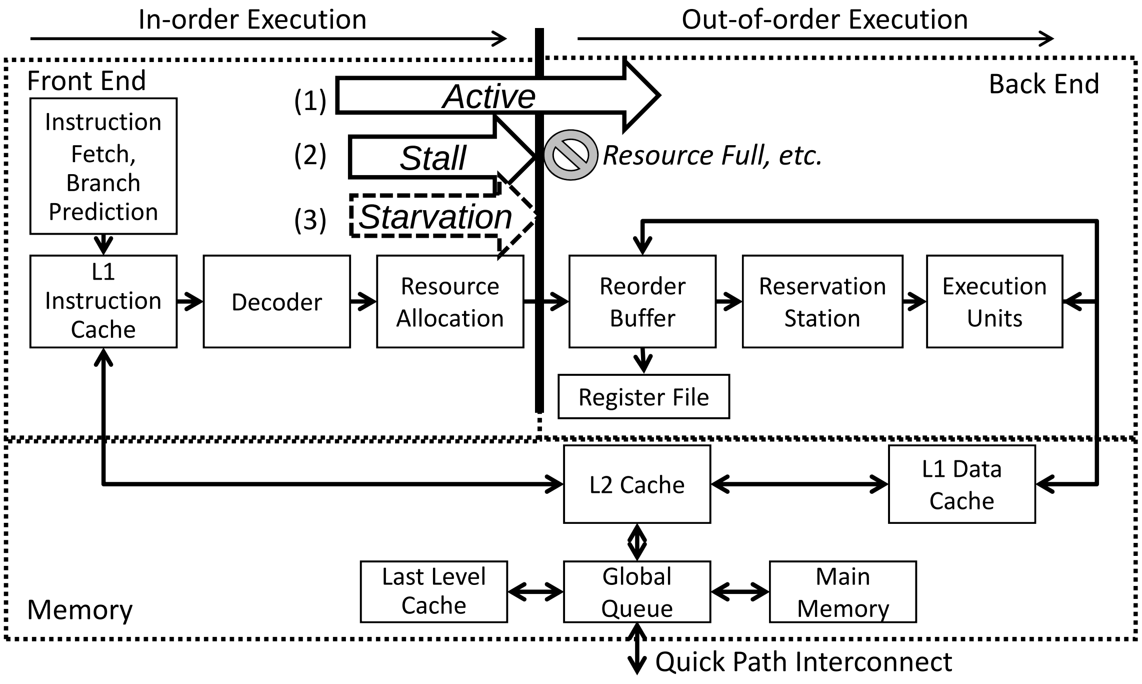


Figure 3.1.: Focus point of the CPU pipeline

The pipeline is composed of a front-end and back-end, as shown in Figure 3.1 [44]. 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 3.1(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 3.1(2)). In addition, we refer to the state in which the μOPs are issued excluding the *instruction-starvation state* and the *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 3.1, Table 3.2 and Table 3.3 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 3.1) 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 [44]. 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:

$$C_{Total} = C_{Active} + C_{Stall} + C_{Starvation} \tag{3.1}$$

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 [45]. 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

$$CPI = CPI0 + \{\sum_{i=2}^{LLC}(H_{Li} \times L_{Li} \times BF_{Li}) + (H_{MM} \times L_{MM} \times BF_{MM})\} \tag{3.2}$$

where LLC denotes last level cache and means the lowest cache in the cache memory hierarchy; the blocking factor [45] is a correction coefficient for concealing the latency by executing

Table 3.1.: Notations for CPU Cost Calculation Model (1)

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 L*i* cache
	LiI	References of instructions to L*i* cache
	MP	Branch mispredictions
	LMM	References to local main memory
	LMMI	References of instructions to local main memory
	LMMD	References of data to local main memory
	RMM	References to remote main memory
	RMMI	References of instructions to remote main memory
	RMMD	References of data to remote 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	L*i* 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 L*i* cache
	LiI	References of instructions to L*i* cache
	LiD	References of data to L*i* cache
	MP	Branch misprediction and instruction cache miss occur simultaneously

Table 3.2.: Notations for CPU Cost Calculation Model (2)

Symbol	Description	
H_{memory}	Ratio of *memory* references to instructions ($H_{memory} = M_{memory}/I$)	
	memory	**Description**
	MM	References to main memory
	Li	References to L*i* cache memory
	LiI	References of instructions to L*i* cache
	LiD	References of data to L*i* 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
	HJ	Hash Join
	Build	Build phase of Hash Join
	Probe	Probe phase of Hash Join
	Cmd	Combination of Hash Join and Nested Loop Join
	CmdBld	Combination build phase
	SMJ	Sort Merge Join
	SortDb	Sorting records in a database table
	SortTmp	Sorting records buffered in a temporary table in main memory
	Merge	Merge Join without sorting
	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
	Total	Total of *ICacheMiss*, *DCacheAcc* and *MP*

Table 3.3.: Notations for CPU Cost Calculation Model (3)

Symbol	Description
$RC_{I\_total(n)}$	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 ($k = 1, 2, \cdots$)
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
P_j	Selectivity of the table j ($j = 0, 1, \cdots$) in SMJ
R_j	Number of records in the table j ($j = 0, 1, \cdots$) in SMJ

instructions in parallel. The second term on the right-hand side of (3.2) 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 (3.2) and the number of instructions I is the pipeline stall cycle (C_{Stall}):

$$C_{Stall} = \sum_{Li=L2}^{LLC} (M_{Li} \times L_{Li} \times BF_{Li}) + (M_{MM} \times L_{MM} \times BF_{MM}) \tag{3.3}$$

$$C_{Total} = CPI \times I = CPI0 \times I + C_{Stall} \tag{3.4}$$

From (3.2)–(3.4), we can show that *CPI0* includes the *active state* and *starvation state*.

$$CPI0 \times I = C_{Active} + C_{Starvation} \tag{3.5}$$

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.

$$C_{Starvation} = C_{ICacheMiss} + M_{MP} \times L_{MP} \times BF_{MP} \tag{3.6}$$

$$C_{ICacheMiss} = \sum_{Li=L2}^{LLC} (M_{LiI} \times L_{Li} \times BF_{LiI}) + (M_{MMI} \times L_{MM} \times BF_{MMI}) \tag{3.7}$$

Here, *BF* is a correction coefficient for considering that both branch misprediction and instruction cache miss occur simultaneously. *ICacheMiss* is expressed as (3.7) by modifying (3.3) because the operations after instruction cache misses and data cache misses are the same. Only the terms relating to branch misprediction are defined.

$$C_{MP} = M_{MP} \times L_{MP} \times BF_{MP} \tag{3.8}$$

According to previous research [46], the CPI of the decision support system benchmark is 1.5 to 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. According to P. Trancoso, *et. al*. [47], the large amount of accessed database data are not reused. 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 [48]. 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 $C_{DCacheAcc}$

$$C_{DCacheAcc} = C_{Active} + C_{Stall} \tag{3.9}$$

$$C_{DCacheAcc} = \sum_{Li=L1}^{LLC} (M_{LiD} \times L_{Li} \times BF_{LiD}) + (M_{MMD} \times L_{MM} \times BF_{MMD}) \tag{3.10}$$

where (3.3) and (3.10) 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

$$C_{Total} = C_{DCacheAcc} + C_{ICacheMiss} + C_{MP} \tag{3.11}$$

In this study, each term on the right-hand side of (3.11) uses statistical information obtained from actual measurements.

In our CPU pipeline activity model, TLB miss penalty is omitted from the cache miss penalty, which is sum of the product of memory latency and the number of memory access. However, the instruction cache miss penalty $C_{ICacheMiss}$ and the data access $C_{DCacheAcc}$ events measured by the CPU performance monitor also include TLB miss penalties. Therefore, in this study, TLB miss penalty is omitted from the cost calculation to simplify the memory access model.

4. DBMS Operation Model

In considering the access path of the query, there are the order of accessing the tables, the order of applying the operators, the order of the joins, and the selection of the join method. Existing cost calculation has been done with a simple linear expression of the number of pages and the number of records. In the cost calculation method of this study, it is possible to create a model as long as processing for cost calculation can be extracted as parts.

In this study, we build a model for selection and join. Since the join method to be selected depends on the selectivity of the table, we consider that the selection operator related to selectivity is integral with join.

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 4.1. 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 inner 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 4.1(a)), which is a 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. Figure 4.1 also shows that HJ is decomposed into a build phase (Figure 4.1(b-1)) and a probe phase (Figure 4.1(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 4.2 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 4.2 (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."

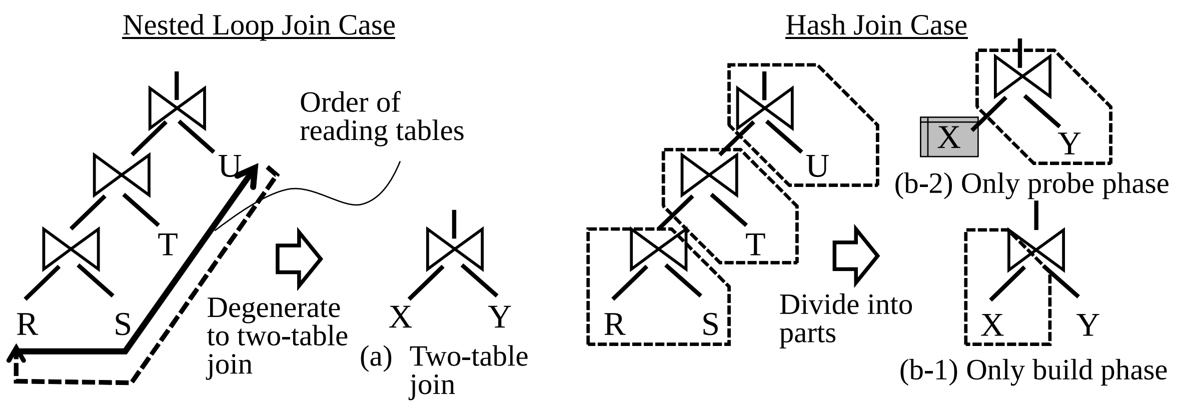


Figure 4.1.: Degradation and Split Cost Calculation Method

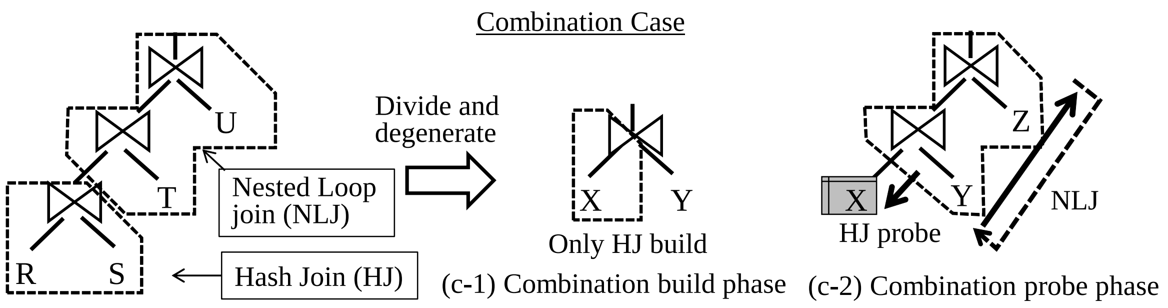


Figure 4.2.: NLJ after HJ Case

The cost of the operation such as grouping, aggregating and sorting (Figure 4.3(d)) except join is divided into two cost calculation parts, storing table scanned or index scanned data into a temporary table on the main memory in Figure 4.3(d-1) and operation such as grouping, aggregating and sorting of the temporary table on the main memory in Figure 4.3(d-2). In particular, the process of Figure 4.3(d-1) includes the selection after the table scan or the index scan. It can be realized by separating CPU statistical information into each processing to divide the statistical information of modeling target operation into several parts. A statistical information is divided into several processes and is the sum of those statistical information in Equation (4.1).

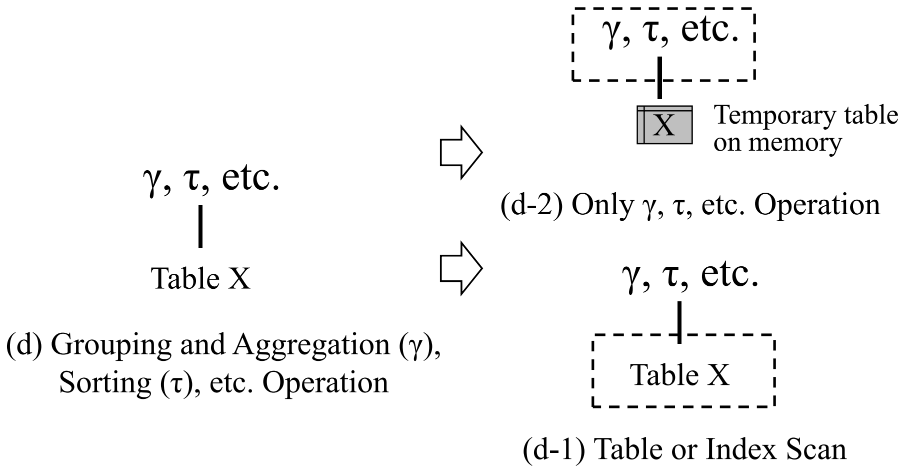

γ, τ, etc.
Table X
(d) Grouping and Aggregation (γ), Sorting (τ), etc. Operation
γ, τ, etc.
X
Temporary table on memory
(d-2) Only γ, τ, etc. Operation
γ, τ, etc.
Table X
(d-1) Table or Index Scan

Figure 4.3.: Split Cost Calculation of Grouping, Aggregating, Sort, etc.

$$\begin{aligned}&\text{(CPU Statistical Information of Operation in Figure 4.3(d-2))}\\&\quad=\text{(CPU Statistical Information of Operation in Figure 4.3(d))}\\&\qquad-\text{(CPU Statistical Information of Operation in Figure 4.3(d-1))}\end{aligned}\tag{4.1}$$

SMJ can be modeled in the same way of HJ mentioned above. SMJ is divided into two type of sort operations (Figure 4.4(e-1) and (e-2)) and a merge join operation (Figure 4.4(e-3)). The sort operation (e-2) sorts records stored in memory temporal table. Those operations (e-1) and (e-2) also filter records according to the condition of `WHERE` clause. The statistical Information of sort operation (e-2) can be obtained by the method in Figure 4.3. The statistical Information of merge join operation (e-3) can be obtained by the method in Figure 4.5. However, it cannot be obtained directly using a simple query as joining two tables by merge join because the query includes the sort operations. Therefore, statistical information on SMJ of two table is introduced in Figure 4.5. The sort operation (e-3) can be calculated using statistical information of (e-1) and the two-table SMJ in Figure 4.5.

Using the above idea, the cost of complex queries can be calculated by accumulating cost parts. For example, the query in Figure 4.6(a) can be divided into parts from (1) to (4). In

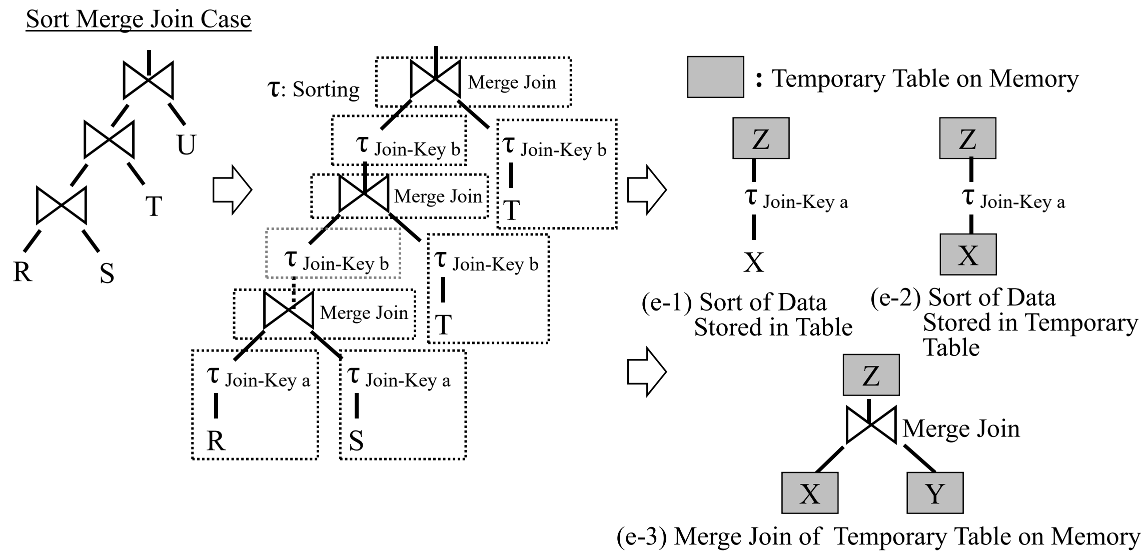

Sort Merge Join Case
U
T
R
S
τ: Sorting
Merge Join
τ Join-Key b
τ Join-Key a
: Temporary Table on Memory
Z
X
Y
(e-1) Sort of Data Stored in Table
(e-2) Sort of Data Stored in Temporary Table
(e-3) Merge Join of Temporary Table on Memory

Figure 4.4.: Sort Merge Join Case

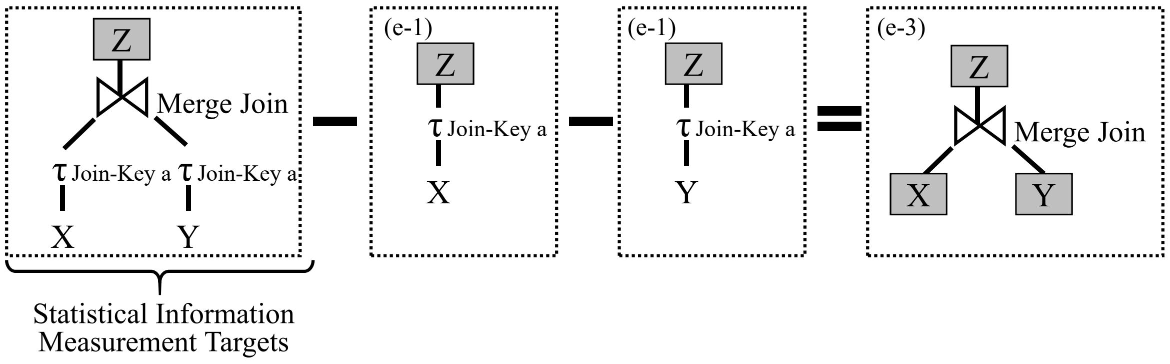


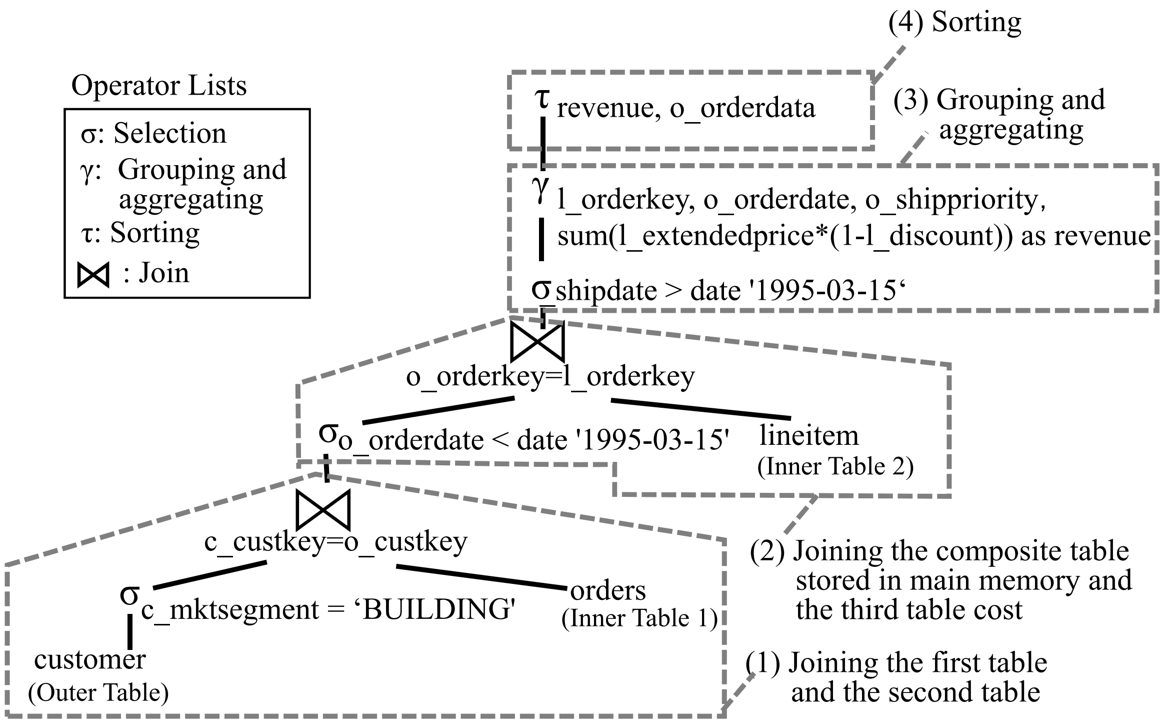
Figure 4.5.: Calculating Method of Statistics information of Sorting Data Stored in Temporary Table for Sort-Merge Join

particular, in the case where three table joins are performed by NLJ, Figure 4.6(1) and (2) can be calculated with one cost formula as shown in Figure 4.1(a). In the join operation (1) and (2), selection operation to the outer table is also included. In the case of HJ, each operation of (1) and (2) can be divided into build phase and probe phase as shown in Figure 4.1(b-1) and (b-2).

As described above, there is a problem that the way of making the cost calculation parts is different depending on the selection of the join method. In this study, we focus on selection of join method as cost calculation target.

```
select
   l_orderkey, sum(l_extendedprice*(1-l_discount)) as revenue, o_orderdate, o_shippriority
from
  customer, orders, lineitem
where
   c_mktsegment = ‘BUILDING'
   and c_custkey = o_custkey
   and l_orderkey = o_orderkey
   and o_orderdate < date '1995-03-15'
   and l_shipdate > date '1995-03-15‘
group by
   l_orderkey, o_orderdate, o_shippriority
order by
  revenue desc, o_orderdata;
```

(a) SQL (TPC-H Query 3)



(4) Sorting
Operator Lists
σ: Selection
γ: Grouping and aggregating
τ: Sorting
⋈ : Join
τ revenue, o_orderdata
(3) Grouping and aggregating
γ l_orderkey, o_orderdate, o_shippriority, sum(l_extendedprice*(1-l_discount)) as revenue
σ shipdate > date '1995-03-15‘
o_orderkey=l_orderkey
σ o_orderdate < date '1995-03-15'
lineitem (Inner Table 2)
c_custkey=o_custkey
(2) Joining the composite table stored in main memory and the third table cost
σ c_mktsegment = ‘BUILDING'
orders (Inner Table 1)
customer (Outer Table)
(1) Joining the first table and the second table

(b) Access Path and Decomposition for Cost Calculation

Figure 4.6.: Dividing TPC-H Query3 [5] into Cost Calculation Parts

5. Cost Calculation Formula

Before considering the cost calculation formulas, we define the inputs and outputs as listed in Table 5.1. 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 the record. Information regarding memory latency and I/O response time is also required. This information can be measured with a simple benchmark program [48].

Table 5.1.: Parameter List for Cost Calculation

Input	Condition of selecting records of table to join, selectivity of tables to join and the number of records of tables
Output	Calculated cost expressed by the number of CPU cycles or by the execution time of the query
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 (3.11) (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 (3.11) for NLJ, HJ, SMJ, and a combination of NLJ and HJ on the condition that the number of inner tables is N, where each element of (3.11) is obtained as a function of the selectivity and number of records in the joining tables. The cost formula of NLJ

$$\begin{aligned} C_{NLJ\_Total}&(P, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \\ &= C_{NLJ\_ICacheMiss}(P, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \\ &\quad + C_{NLJ\_MP}(P, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \\ &\quad + C_{NLJ\_DCacheAcc}(P, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \end{aligned} \tag{5.1}$$

is obtained by combining (3.7), (3.8), (3.10), and (3.11). The cost related to each element of the instruction cache miss, branch misprediction, and data reference are expressed as

$$\begin{aligned}C_{NLJ\_ICacheMiss}&(P, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN})\\ &= M_{L2I}(P, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \times L_{L2} \times BF_{L2I}\\ &+ M_{LLCI}(P, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \times L_{LLC} \times BF_{LLCI}\\ &+ M_{MMI}(P, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \times L_{MM} \times BF_{MMI}\end{aligned} \tag{5.2}$$

$$\begin{aligned}C_{NLJ\_MP}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) &= M_{MP}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN})\\ &\times L_{MP} \times BF_{MP}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN})\end{aligned} \tag{5.3}$$

$$\begin{aligned}&C_{NLJ\_DCacheAcc}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN})\\ &= M_{L1D}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \times L_{L1} \times BF_{L2D}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN})\\ &+ M_{L2D}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \times L_{L2} \times BF_{L2D}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN})\\ &+ M_{LLCD}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \times L_{LLC} \times BF_{LLCD}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN})\\ &+ M_{MMD}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \times L_{MM} \times BF_{MM}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}).\end{aligned} \tag{5.4}$$

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 as follows.

$$C_{HJ}(P, R) = C_{Build\_Total}(P, R) + \sum_{k=1}^{N} C_{Probe\_Total}(P, R) \times \left\lceil \frac{\mathit{Total\ Size\ of\ Hash\ Table}}{\mathit{Size\ of\ Join\ Buffer}} \right\rceil \tag{5.5}$$

$$C_{Phase\_Total}(P, R) = C_{Phase\_ICacheMiss}(P, R) + C_{Phase\_MP}(P, R) + C_{Phase\_DCacheAcc}(P, R) \tag{5.6}$$

$$\begin{aligned}C_{Phase\_ICacheMiss}&(P, R)\\ &= M_{L2I}(P, R) \times L_{L2} \times BF_{L2I}(P, R) + M_{LLCI}(P, R) \times L_{LLC} \times BF_{LLCI}(P, R)\\ &+ M_{MMI}(P, R) \times L_{MM} \times BF_{MMI}(P, R)\end{aligned} \tag{5.7}$$

$$C_{Phase\_MP}(P,R) = M_{MP}(P,R) \times L_{MP} \times BF_{MP}(P,R) \tag{5.8}$$

$$\begin{aligned} &C_{Phase\_DCacheAcc}(P,R) \\ &\quad = M_{L1D}(P,R) \times L_{L1} \times BF_{L2D}(P,R) + M_{L2D}(P,R) \times L_{L2} \times BF_{L2D}(P,R) \\ &\quad + M_{LLCD}(P,R) \times L_{LLC} \times BF_{LLCD}(P,R) + M_{MMD}(P,R) \times L_{MM} \times BF_{MMD}(P,R) \end{aligned} \tag{5.9}$$

where

$$\{Phase, P, R\} = \begin{cases} \{Build, P_O, R_O\} & \text{build phase} \\ \{Probe, P_{Ik}, R_{Ik}\}(k = 1, 2, \cdots, N) & \text{probe phase} \end{cases}$$

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 4.2(c-1), the cost formula of the combination build phase can be created with reference to the cost formula of the HJ build phase.

$$\begin{aligned} &C_{Cmb\_Total}(P_O, P_{Ik}, R_O, R_{Ik}) = C_{CmbBld\_Total}(P_O, R_O) \\ &\quad + C_{CmbPrb\_Total}(P_O, P_{Ik}, R_O, R_{Ik}) \times \left\lceil \frac{Total\ Size\ of\ Hash\ Table}{Size\ of\ Join\ Buffer} \right\rceil \quad (k = 1, 2, \cdots, N) \end{aligned} \tag{5.10}$$

$$\begin{aligned} &C_{CmbBld\_Total}(P_O, R_O) \\ &\quad = C_{CmbBld\_ICacheMiss}(P_O, R_O) + C_{CmbBld\_MP}(P_O, R_O) \\ &\quad + C_{CmbBld\_DCacheAcc}(P_O, R_O) \end{aligned} \tag{5.11}$$

$$\begin{aligned} &C_{CmbBld\_ICacheMiss}(P_O, R_O) \\ &\quad = M_{L2I}(P_O, R_O) \times L_{L2} \times BF_{L2I}(P_O, R_O) + M_{LLCI}(P_O, R_O) \times L_{LLC} \times BF_{LLCI}(P_O, R_O) \\ &\quad + M_{MMI}(P_O, R_O) \times L_{MM} \times BF_{MMI}(P_O, R_O) \end{aligned} \tag{5.12}$$

$$C_{CmbBld\_MP}(P_O, R_O) = M_{MP}(P_O, R_O) \times L_{MP} \times BF_{MP}(P_O, R_O) \tag{5.13}$$

$$\begin{aligned} &C_{CmbBld\_DCacheAcc}(P_O, R_O) \\ &\quad = M_{L1D}(P_O, R_O) \times L_{L1} \times BF_{L2D}(P_O, R_O) \\ &\quad + M_{L2D}(P_O, R_O) \times L_{L2} \times BF_{L2D}(P_O, R_O) + M_{LLCD}(P_O, R_O) \times L_{LLC} \times BF_{LLCD}(P_O, R_O) \\ &\quad + M_{MMD}(P_O, R_O) \times L_{MM} \times BF_{MMD}(P_O, R_O) \end{aligned} \tag{5.14}$$

$$\begin{aligned}
&C_{CmbPrb\_Total}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \\
&\quad = C_{CmbPrb\_ICacheMiss}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \\
&\quad + C_{CmbPrb\_MP}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \\
&\quad + C_{CmbPrb\_DCacheAcc}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \qquad (5.15)
\end{aligned}$$

For a combination case such as Figure 4.2(c-2), the cost formula of the combination probe phase can be created with reference to the cost formula of NLJ.

$$\begin{aligned}
&C_{CmbPrb\_ICacheMiss}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \\
&\quad = M_{L2I}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \times L_{L2} \times BF_{L2I} \\
&\quad + M_{LLCI}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \times L_{LLC} \times BF_{LLCI} \\
&\quad + M_{MMI}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \times L_{MM} \times BF_{MMI} \qquad (5.16)
\end{aligned}$$

$$\begin{aligned}
&C_{CmbPrb\_MP}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \\
&\quad = M_{MP}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \times L_{MP} \times BF_{MP}(P_O, R_O, R_{Ik}) \qquad (5.17)
\end{aligned}$$

$$\begin{aligned}
&C_{CmbPrb\_DCacheAcc}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \\
&= M_{L1D}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \times L_{L1} \times BF_{L2D}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \\
&\quad + M_{L2D}(P, R_O, R_{Ik}) \times L_{L2} \times BF_{L2D}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \\
&\quad + M_{LLCD}(P, R_O, R_{Ik}) \times L_{LLC} \times BF_{LLCD}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \\
&\quad + M_{MMD}(P, R_O, R_{Ik}) \times L_{MM} \times BF_{MM}(P_O, P_{I1}, \cdots, P_{IN}, R_O, R_{I1}, \cdots, R_{IN}) \qquad (5.18)
\end{aligned}$$

Sort Merge Join (SMJ) operation is divided into three type of operations, sorting data from a table of database in Figure 4.4(e-1), sorting data from a temporary table in main memory in Figure 4.4(e-2) and merge join in Figure 4.4(e-3). The cost of those operations is defined in Table 5.2. The cost of whole SMJ is defined as C_{SMJ_Total}.

Input parameters of each cost calculation formulas in SMJ operations are explained using three tables join case in Figure 5.1. First, the operation (a), (b) and (c) are selection of records with probability P_0, P_1 and P_2 from the tables, which have R_0, R_1 and R_2 records and sorting those records respectively. The sort operation costs are given by $C_{SortDb\_Total}(R_0, P_0)$, $C_{SortDb\_Total}(R_1, P_1)$ and $C_{SortDb\_Total}(R_2, P_2)$. Next, the merge join operation (d) join the output records from sorting operation (a) and (b). Therefore, the operation cost is given by $C_{Merge\_Total}(R_0, R_1, P_0, P_1)$. Then, the sorting operation (e) sort the outputs from operation (d). The cost is given by $C_{SortTmp\_Total}(R_0, R_1, P_0, P_1)$. After that, the merge join operation (f) join the outputs of the operation (c) and (e). The cost is given by $C_{Merge\_Total}(R_0, R_1, R_2, P_0, P_1, P_2)$. Finally, total cost is given by the following equation.

Table 5.2.: Parts of SMJ and Notation of Cost

Operation (Figure 4.4)	Notation of Cost	Input Parameters
Sorting data from a table of database (e-1)	C_{SortDb_Total}	Number of Records of a table and selectivity
Sorting data from a temporary table in main memory (e-2)	$C_{SortTmp\_Total}$	Number of Records of temporary table or buffer for join in main memory and selectivity of that records
Merge join (e-3)	C_{Merge_Total}	Number of records in tables to be joined and selectivity of those records

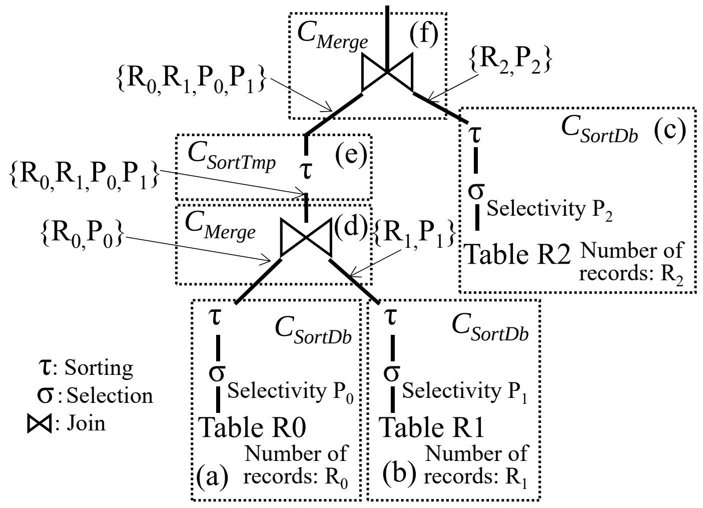


Figure 5.1.: Input Parameters to Each Part of Sort Merge Join

$$C_{SMJ\_Total}(P_0, P_1, P_2, R_0, R_1, R_2) = \sum_{j=0}^{2} C_{SortDb\_Total}(P_0, \cdots, P_j, R_0, \cdots, R_j)$$
$$+ C_{SordTmp_{Total}}(P_0, P_1, R_0, R_1) + \sum_{j=0}^{1} C_{Merge_{Total}}(P_0, \cdots, P_j, R_0, \cdots, R_j) \quad (5.19)$$

The generalized SMJ cost calculation formula is given by equation 5.20.

$$C_{SMJ\_Total}(P_0, \cdots, P_{N-1}, R_0, \cdots, R_{N-1}) = \sum_{j=0}^{N-1} C_{SortDb\_Total}(P_j, R_j)$$
$$+ \sum_{j=1}^{N-2} C_{SordTmp_{Total}}(P_0, \cdots, P_j, R_0, \cdots, R_j) + \sum_{j=1}^{N-1} C_{Merge_{Total}}(P_0, \cdots, P_j, R_0, \cdots, R_j) \quad (5.20)$$

where N is the number of tables to join.

Each part of SMJ operations are obtained using CPU events, instruction cache miss penalty, branch misprediction penalty and data cache access time as following equations 5.21, 5.22, 5.23 and 5.24.

$$C_{Phase\_Total}(\boldsymbol{P}, \boldsymbol{R}) = C_{Phase\_ICacheMiss}(\boldsymbol{P}, \boldsymbol{R}) + C_{Phase\_MP}(\boldsymbol{P}, \boldsymbol{R}) + C_{Phase\_DCacheAcc}(\boldsymbol{P}, \boldsymbol{R}) \quad (5.21)$$

$$C_{Phase\_ICacheMiss}(\boldsymbol{P}, \boldsymbol{R})$$
$$= M_{L2I}(\boldsymbol{P}, \boldsymbol{R}) \times L_{L2} \times BF_{L2I}(\boldsymbol{P}, \boldsymbol{R}) + M_{LLCI}(\boldsymbol{P}, \boldsymbol{R}) \times L_{LLC} \times BF_{LLCI}(\boldsymbol{P}, \boldsymbol{R})$$
$$+ M_{MMI}(\boldsymbol{P}, \boldsymbol{R}) \times L_{MM} \times BF_{MMI}(\boldsymbol{P}, \boldsymbol{R}) \quad (5.22)$$

$$C_{Phase\_MP}(\boldsymbol{P}, \boldsymbol{R}) = M_{MP}(\boldsymbol{P}, \boldsymbol{R}) \times L_{MP} \times BF_{MP}(\boldsymbol{P}, \boldsymbol{R}) \quad (5.23)$$

$$C_{Phase\_DCacheAcc}(\boldsymbol{P}, \boldsymbol{R})$$
$$= M_{L1D}(\boldsymbol{P}, \boldsymbol{R}) \times L_{L1} \times BF_{L2D}(\boldsymbol{P}, \boldsymbol{R}) + M_{L2D}(\boldsymbol{P}, \boldsymbol{R}) \times L_{L2} \times BF_{L2D}(\boldsymbol{P}, \boldsymbol{R})$$
$$+ M_{LLCD}(\boldsymbol{P}, \boldsymbol{R}) \times L_{LLC} \times BF_{LLCD}(\boldsymbol{P}, \boldsymbol{R}) + M_{MMD}(\boldsymbol{P}, \boldsymbol{R}) \times L_{MM} \times BF_{MMD}(\boldsymbol{P}, \boldsymbol{R}) \quad (5.24)$$

where

$$\{Phase, \boldsymbol{P}, \boldsymbol{R}\} = \begin{cases} \{SordDb, P_j, R_j\} & \text{Sorting records in a database table} \\ \{SordTmp, P_0, \cdots, P_j, R_0, \cdots, R_j\} & \\ & \text{Sorting records buffered in a temporary table} \\ \{Merge, P_0, \cdots, P_j, R_0, \cdots, R_j\} & \text{Merge Join} \end{cases}$$

and $j = 0, 1, \cdots, N-1$.

The accuracy of cost depends on the accuracy of CPU statistical information such as the number of cache memory accesses, the number of main memory accesses and blocking factors. It is important to choose how to obtain those parameters. We compare the technologies from the viewpoint of estimating the execution time.

We can choose a method out of three methods, actual measurement using performance monitor embedded CPU when executing query, modeling the behavior of query execution and simulation of query execution as shown in Table 5.3. The method of modeling the behavior of query execution [25] has high versatility. That is because it is not influenced by the difference of hardware and software and is not necessary to measure statistics information. However, it is very difficult to model instruction cache activity because it is difficult to find when instruction cache misses happen. The simulation of query execution is easy to reflect architectural change of different generation CPU to simulation models; however, accurate emulation of CPU increase calculation time [49]. From the viewpoint of accuracy and time to obtain parameters, we chose a method to use the performance monitor which gives the most accurate parameters.

Table 5.3.: Parameter Estimation Methods

Method of obtaining Parameters	**Advantage**	**Disadvantage**
Actual measurement using performance monitor embedded CPU	Acquiring accurate statistical information of CPU	Updating parameters when CPU architecture is changed
Modeling the behavior of query execution	Small influence of difference in CPU architecture and version of software	Difficult to model instruction cache statistics
Simulation of query execution		Increasing calculation time by accurate emulation

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, which CPU socket the main memory modules are installed in, and other factors. According to J. L. Lo *et al*. [50], the memory access concentration is low when executing analytic queries, such as the TPC-H benchmark, and does not increase the memory latency.

6. Feasibility Study of Creating Cost Calculation Formulas

In this study, we propose a method for obtaining cost calculation formulas for join operations using statistical information measured by the performance monitor on a CPU. In this chapter, we use a comparatively large database for a feasibility study of the proposed method and obtain cost formulas of nested loop join (NLJ) and hash join (HJ) of two tables. The cross point of these cost calculation formulas is obtained and its prediction accuracy is reported. First, we describe the measurement environment, and subsequently demonstrate the cost calculation formulas obtained from the measurement results. Finally, we evaluate the accuracy of the cross point obtained from the cost formulas. Cost calculation formulas for sort merge join (SMJ) can be obtained with the same approach. Since there is no SMJ support in the DBMS used in this experiment, only NLJ and HJ are targeted here.

6.1. CPU Events Measurement for Parameter Setting of Cost Calculation Formulas

CPU statistical information, such as the numbers of instructions and cache hits, is measured in the environment in Table 6.1. The parameters in Table 5.1 are calculated using this information. The server has two CPU sockets with non-uniform memory access (NUMA). The main memory of NUMA is composed of the local memory attached to CPU directly and remote memory attached to the other CPU connected through the interconnection network between CPUs. NVMe Flash SSD is used as storage for storing the database. We used high-speed NVMe SSD to reduce the time taken for the disk I/O as much as possible and to obtain an environment closer to the in-memory database environment.

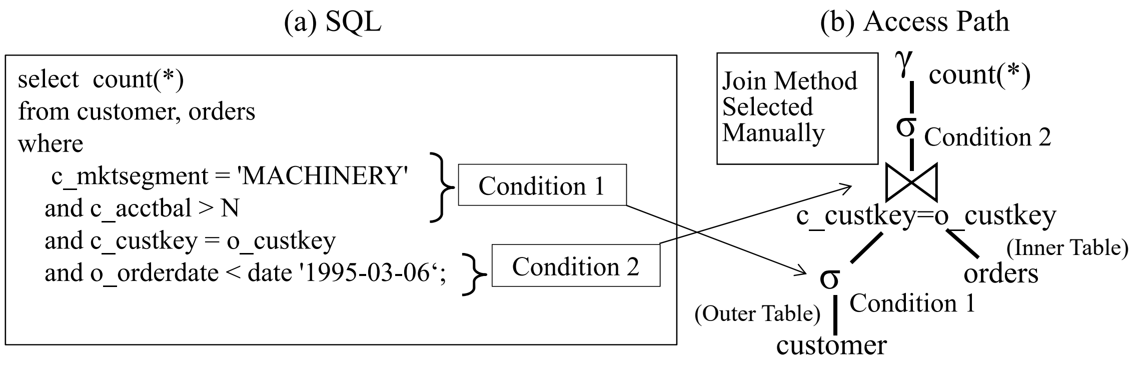
We used open-source MariaDB [51]. MariaDB supports multithreaded, asynchronous I/O as it utilizes the latest hardware characteristics and it supports multiple join methods, including NLJ and HJ. The NLJ supported by MariaDB is block nested loop join (BNL), which improved the NLJ, but under the conditions of the query and index used in this study, it operates in the same way as the general NLJ. Moreover, as MariaDB does not support NUMA, we set interleave using OS startup parameters. The MariaDB version used in this study does not function effectively for automatically selecting a join method based on the cost it calculates, and fixedly selects the join method with user-configurable parameters.

The evaluated query and the measurement condition are shown in Figure 6.1. We used the modified *query 3* of TPC-H benchmark [5] for the evaluation of two-table join, and obtained a description by extracting only join processing. SF100 (100 GB) of TPC-H was used as

Table 6.1.: Measurement Environment for Feasibility Study

CPU	Xeon L5630 2.13GHz Quad-core, LLC 12MB [Westmere-EP] ×2
Memory	DDR3 24GB (4GB ×6)
Disk (DB)	PCIe NVMe Flash SSD 800GB ×1 (Note: Max throughput suppressed by server's PCIe I/F (Ver.1.0a), approximately 1/4 of max throughput.)
Disk (OS)	SAS 10krpm 600GB, RAID5 (4 Data + 1 Parity)
OS	CentOS 6.6 (x64)
DBMS	MariaDB 10.1.8

a database. The number of records in the outer table and inner table referenced at the join changes the selection rate of the data to be referenced by changing the search condition of the query against the c_acctbal column of the inner table (Figure 6.1(c)). For NLJ, by changing the number of rows in the inner table, the number of records in the inner table corresponding to the key to be joined specified in the outer table was changed.(Figure 6.1(d)) The index of the database is set as the primary key defined according to the specification of TPC-H [5]. The CPU performance counter data were measured using Intel® Vtune™ Amplifier XE. For understanding the mean of counters, we refer to the literature [44]. The counter measured mainly collects information related to the reference to the cache memory and the state of the pipeline such as the number of stall cycles.



```
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 (Condition 1)

N	9998	9978	9798	9200	9000	8000	7000
Selectivity	3.62E-05	4.00E-04	3.67E-03	1.45E-02	1.82E-02	3.64E-02	5.45E-02

(d) Number of Inner Table Records (Condition2)

Number of Records	150,000,000	112,500,000	75,000,000	37,500,000

Figure 6.1.: Target Query of Measurement and Cost Estimation

6.2. Measurement Results and Cost Calculation Formulas

In this section, the measurement results of NLJ and HJ are presented. As the number of rows referenced by NLJ increases in proportion to the selectivity, it is presumed that the number of executed instructions and the number of memory references will increase. In addition, as the amount of referenced data increases, it is assumed that the data in the cache memory are replaced with new data frequently and the hit ratio becomes lower. Based on these presumptions, we will analyze the measurement results by focusing on model creation via linear regression.

In the build phase of HJ, regardless of the selectivity of the outer table, all records are accessed; therefore, it is presumed that the number of executed instructions and the number of memory references are constant with respect to the selectivity. Even in the probe phase of HJ, it is presumed that there is a similar tendency because all the records in the inner table are accessed. However, as the amount of data stored in the hash table constructed in the build phase increases in proportion to the selectivity, it is presumed that the tendencies of the number of executed instructions and the number of memory references are similar to those in NLJ. We analyze the measurement results based on the above presumptions.

6.2.1. Measurement Results of NLJ

Figure 6.2 shows the relationship between the selectivity of the outer table and the number of executed instructions when joining two tables. The number of executed instructions increases almost linearly with respect to the selectivity. Even if the number of records in the inner table is changed, it has the same tendency as the executed instructions. The straight line is a regression line. The coefficient of determination (R^2), which is an index representing the goodness of approximation, is one. This indicates that the regression line can be approximated with high accuracy. Regression analysis of the slope and intercept of the regression line of the number of executed instructions against the number of lines in the inner table shows that the coefficient of determination is approximately one and it can be observed that the number of lines in the inner table can be approximated by a straight line (Figure 6.2(b), (c)).

Figure 6.3(a) shows the relationship between the number of executed instructions and the hit ratio of the L1I cache. The measured values classified by the number of rows in the inner table have shapes such as hyperbolas. Assuming that the measured values can be approximated by a hyperbolic curve from the shape of the distribution, they are analyzed using the concept shown in Figure 6.4.

First, we introduced a dummy variable corresponding to the asymptotic line and take logarithms on both sides of the assumed hyperbolic equation to replace the variable. When this process was applied to the measured values, it was observed that the graph in Figure 6.3(b) was obtained, and as the coefficient of determination of linear regression was one, it could be approximated by a straight line. Therefore, the number of executed instructions and the L1I hit rate can be approximated by hyperbolic curves.

Subsequently, we analyzed the graph of slope and intercept of the regression line in Figure 6.3 (b) by hyperbolic approximation similarly. As shown in Figure 6.3 (c), the slope, intercept, and dummy variable (dm1) can be approximated by straight lines. Therefore, they can also be modeled by hyperbolic functions.

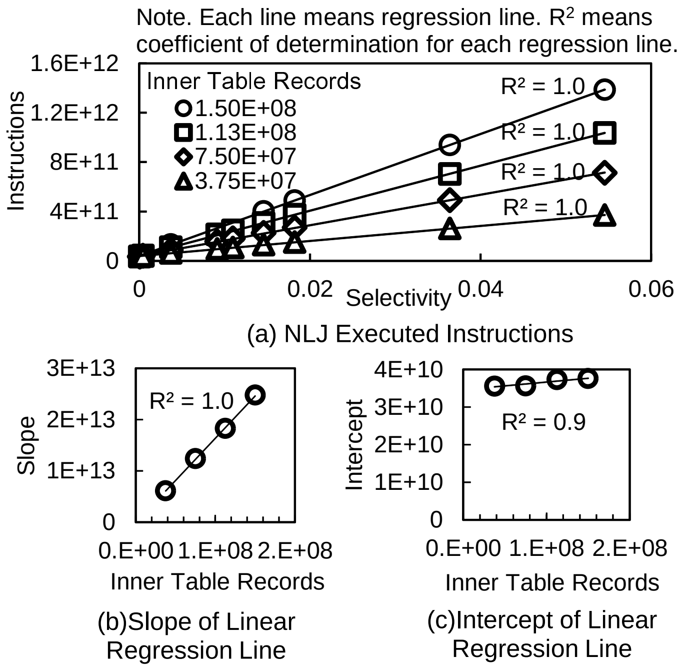


Figure 6.2.: Number of Instructions on NLJ and Regression Line

Figure 6.5(a) shows the relationship between the L1I cache miss ratio and the L2 cache hit ratio limited to the instruction references. It can be observed that the L1I cache miss ratio and the L2 cache hit ratio limited to the executed instructions can be linearly approximated from the shape of the graph and the value of the decision coefficient obtained via regression analysis.

Furthermore, the slope and intercept of the linear approximation formula obtained for each number of records in the inner table can be approximated by a straight line as shown in the graph on the right side of Figure 6.5(a).

Therefore, the formula to calculate the L2 cache hit ratio from the L1 cache miss ratio can be obtained by substituting each linear approximation formula of the slope and intercept, which is a function of the number of records of the inner table on the right side of Figure 6.5(a), for the slope and intercept.

Subsequently, we describe the tendency of references to last-level cache (LLC) and main memory. The LLC and main memory installed in the CPU socket on which the DBMS is running are called local LLC and local main memory, respectively. The LLC and main memory installed in the other CPU socket on which the DBMS is not running are called remote LLC and remote main memory, respectively. The tendency of accesses to LLC and main memory is shown in Figure 6.5(b)–(e). The local LLC hit ratio can be approximated by a straight line with respect to the L2 miss ratio. The Remote LLC hit ratio, the local main memory access ratio and the remote cache access ratio have the same tendency as the local LLC. On the other hand, Regression analysis of the slope and intercept of the linear

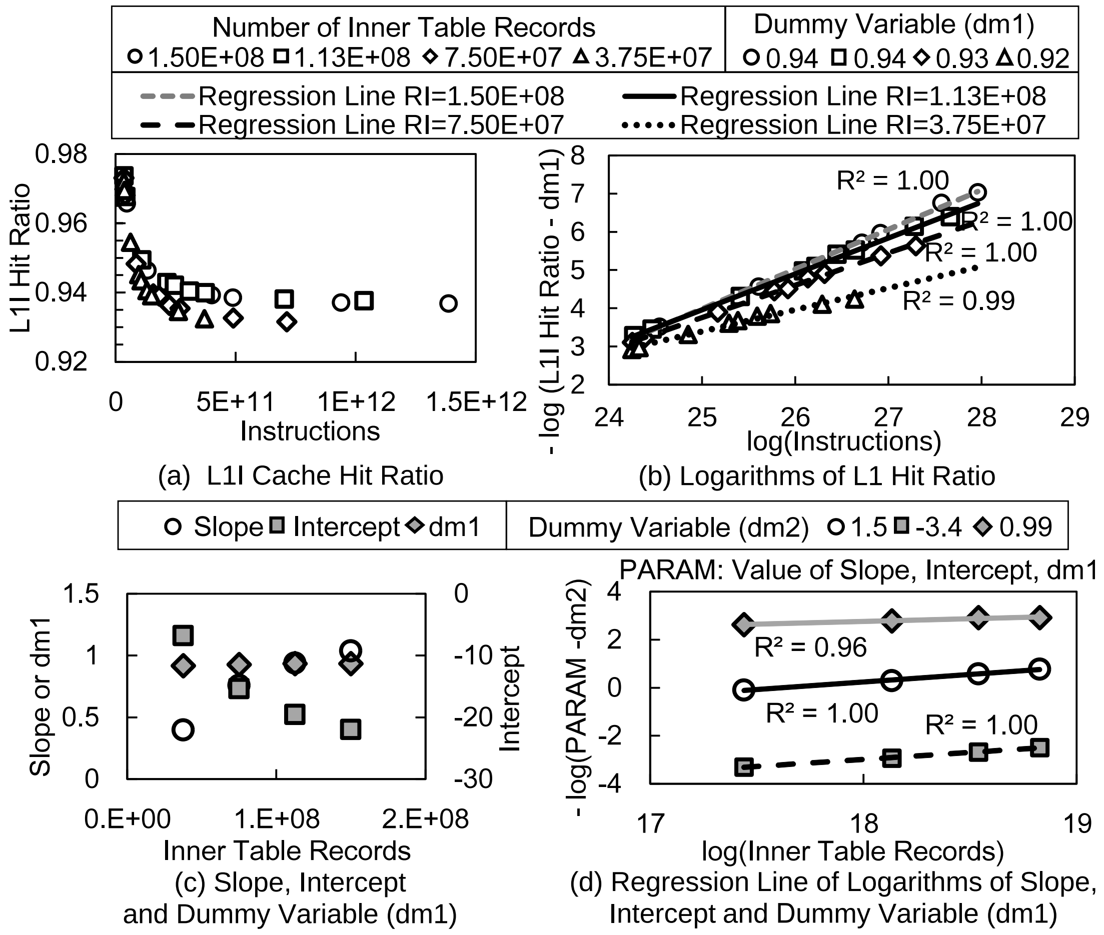


Figure 6.3.: Number of Instructions and L1I Hit Ratio on NLJ

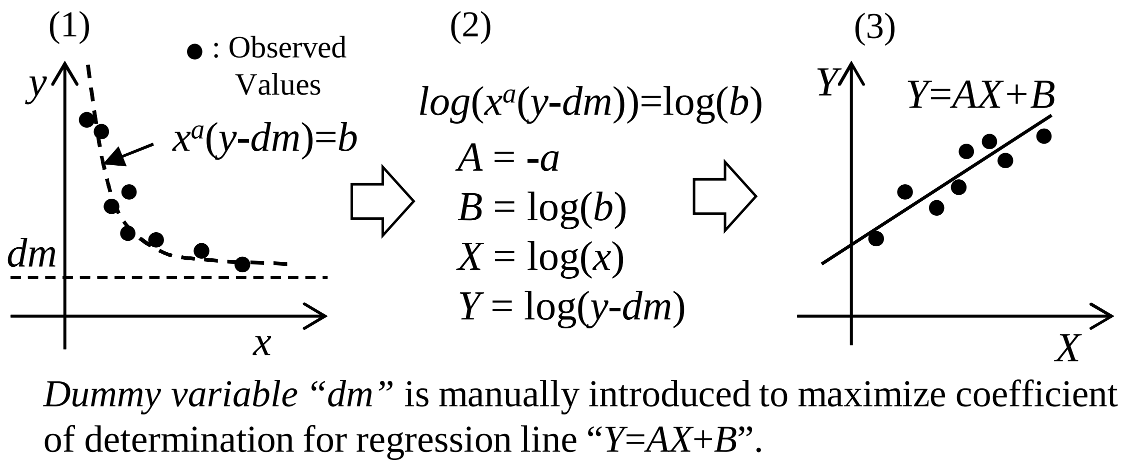

(1)
● : Observed Values
$x^a(y\text{-}dm)=b$
y
dm
x
(2)
$log(x^a(y\text{-}dm))=\log(b)$
$A = -a$
$B = \log(b)$
$X = \log(x)$
$Y = \log(y\text{-}dm)$
(3)
$Y=AX+B$
Y
X
Dummy variable "dm" is manually introduced to maximize coefficient of determination for regression line "*Y=AX+B*".

Figure 6.4.: Estimation Method of Curves Using Dummy Variable

approximation formula obtained for each number of records in the inner table reveals that the determination coefficient of the intercept is approximately 0.3 and it cannot be said to be approximated linearly.

Similarly, the remote LLC hit ratio and the local main memory reference rate (the ratio of instructions obtained from the local main memory to all executed instructions) can be approximated by a regression line with respect to the local LLC miss rate. However, when the regression analysis is performed on the slope and intercept of the linearly approximated expression against the number of rows in the inner table, the coefficient of determination is small and it is difficult to conclude that it can be approximated by a straight line.

Therefore, in this study, it is determined whether it can be approximated by a straight line with the magnitude of the decision coefficient, but as the criterion for determining whether it can be approximated by a straight line varies from one study to another, a tentative intermediate value of 0.5 is used as a criterion. Thus, modeling with a linear function is performed in the cases where the coefficient of determination is 0.5 or more, and the average measured values are used in the cases where the coefficient of determination is less than this value. L2, LLC, and main memory access case described above can be approximated by a straight line because those coefficients of determination are more than 0.5 in Table 6.2.

Table 6.2.: Coefficient of Determination of Regression Line of Instruction Access on NLJ

	Records of Inner Table			
	1.50×10^8	1.13×10^8	7.50×10^7	3.75×10^7
L2	1.00	1.00	1.00	1.00
Local LLC	1.00	1.00	1.00	1.00
Remote LLC	1.00	0.97	0.96	1.00
Local Main Memory	0.84	0.90	0.95	0.64
Remote Main Memory	1.00	0.94	0.96	0.99

Subsequently, the measurement results related to data cache are described. The measurement results of load instructions obtained for each number of records in the inner table can be linearly approximated as shown in Figure 6.6(a). In addition, it can be observed that the slope and intercept of these linear approximation equations can be linearly approximated to the number of records of the inner table. Similarly, in the cases of Figure 6.6(b) to (f), the results of the regression analysis of the measurement values on the left side have a coefficient of determination of 0.99 and can be approximated by a straight line as shown in Figure 6.6(a). The slope and intercept of these approximate lines were analyzed for the number of rows in the inner table, as shown in the graph on the right side of Figure 6.6(b) to (f), where it was observed that some slopes and intercepts are difficult to approximate with straight lines.

As the measurement results of L1D hit case have a hyperbolic distribution, the measurement results are converted into logarithms like an L1I hit to create a graph (Figure 6.7(a)). These measurement results are distributed on a straight line, but the slope and intercept are varied, and the coefficient of determination is less than 0.5 (Figure 6.7(b)). Therefore, the average

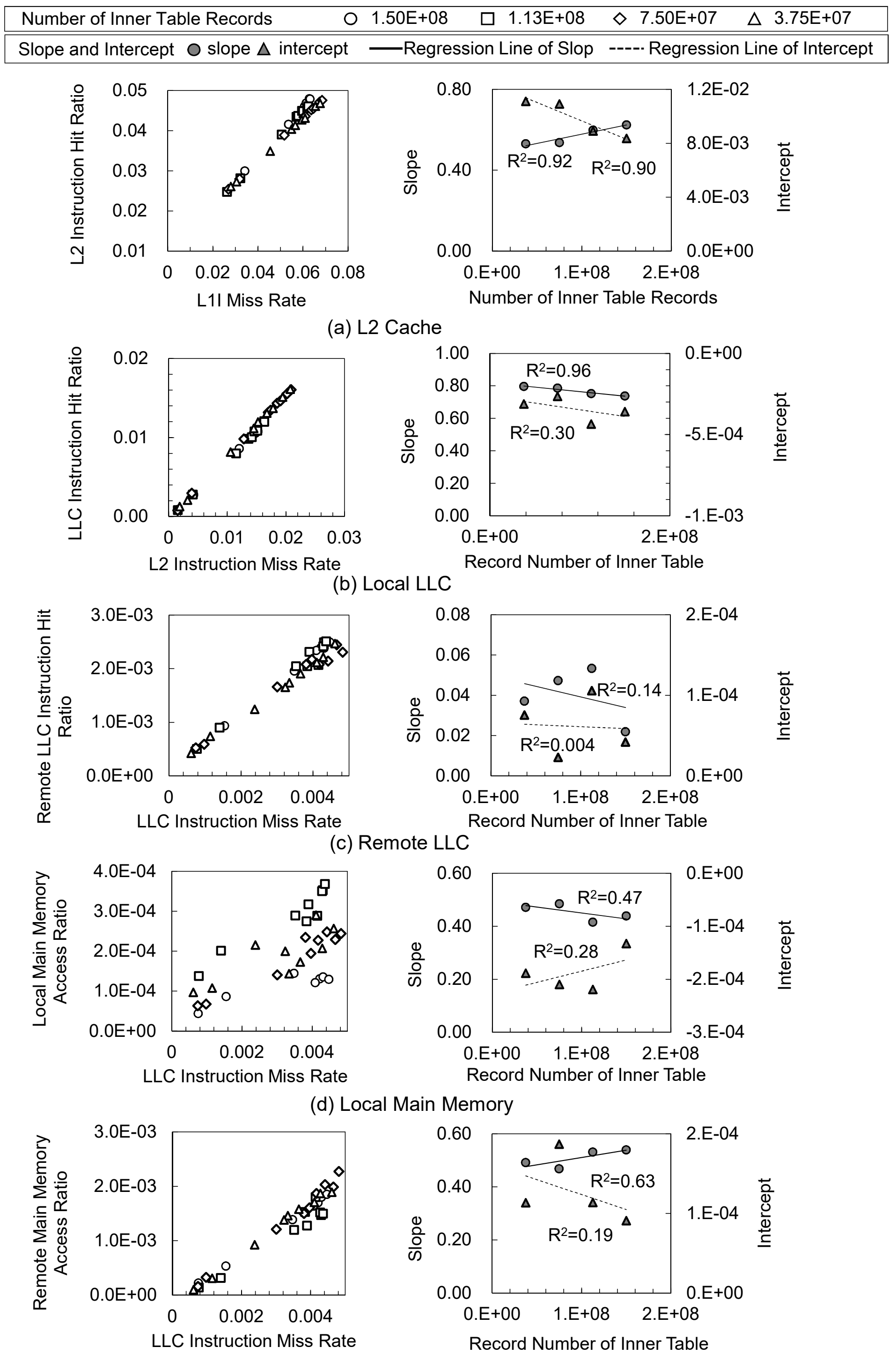


Figure 6.5.: Instruction Access Ratio of L2, LLC and Main Memory on NLJ

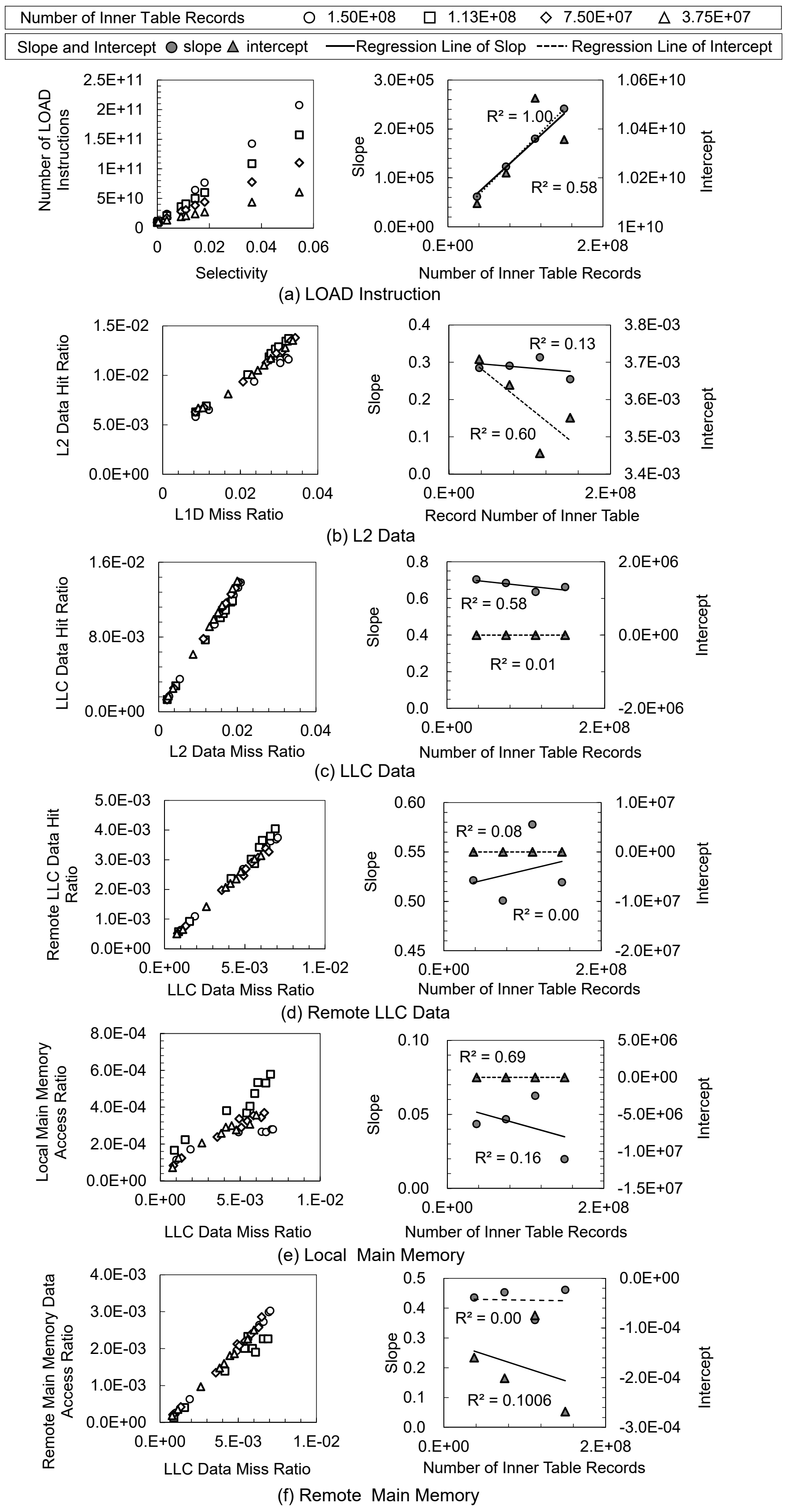


Figure 6.6.: Data Access Ratio of L2, LLC and Main Memory on NLJ

values of slope, intercept, and dummy variable are used as it is difficult to determine whether they can be approximated by linear regression.

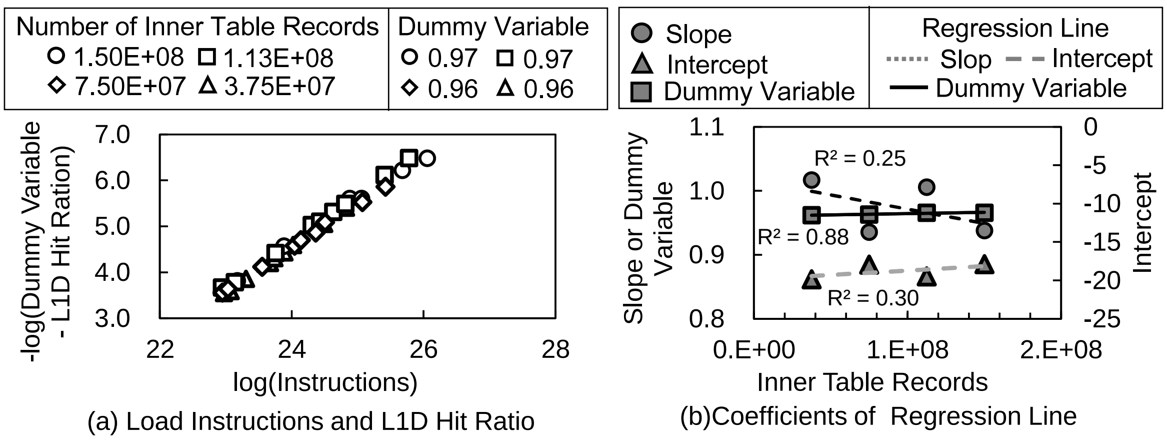


Figure 6.7.: Relation between Number of Inner Table Records and L1D Hit Ratio on NLJ

Figures 6.8(a) and (b) show the relationship between the sum of the product of the number of instructions or data references and memory latency for each memory layer, i.e., cache miss penalty and $C_{ICacheMiss}$ or $C_{DCacheAcc}$. In both cases, as the coefficient of determination obtained via regression analysis is one, it can be approximated by a straight line. Figures 6.10(a) and (b) show the relationship between the slopes of these straight lines and the number of records of the inner table. We also model them using the decision criteria based on the decision coefficient.

For branch misprediction, as shown in Figure 6.9, it can be observed that a straight line passing through the origin can be used for approximation. Regarding the slope of this regression line, by analyzing the relation to the number of records in the inner table, the determination coefficient is 0.98 as shown in Figure 6.10(c). Therefore, the measurement results of branch misprediction can be approximated by a straight line.

From the above discussion, Equations (6.1), (6.2), and (6.3) are obtained. The definitions of parameters of the equations are listed in Table 6.3.

$$C_{NLJ\_ICacheMiss} = I \times (H_{L2I} \times L_{L2} + H_{LLLCI} \times L_{LLLCI} + H_{RLLC} \times L_{RLLC} + H_{LMMI} \times L_{LMM} + H_{RMMI} \times L_{RMM}) \times (K0_1 \times R_I + S0_1) + A0_1 \qquad (6.1)$$

where

$$I = (K0_2 \times R_I + S0_2) \times R_O \times P + (K0_3 \times R_I + S0_3)$$

$$H_{L1I} = e^{-S0_4 \times R_I^{-K0^4} - d0} \times I^{e^{-S0_5 \times R_I^{-K0_5} - d1}} - e^{S0_6} \times R_I^{-K0_6} + d2$$

$$H_{L2I} = (K0_7 \times R_I + S0_7) \times (1 - H_{L1I}) + (K0_8 \times R_I + S0_8)$$

$$H_{LLLCI} = (K0_9 \times R_I + S0_9) \times (1 - H_{L1I} - H_{L2I}) + A0_2$$

$$H_{RLLCI} = A0_3 \times (1 - H_{L1I} - H_{L2I}) + A0_4$$

$$H_{LMMI} = A0_5 \times (1 - H_{L1I} - H_{L2I}) + A0_6$$

$$H_{RMMI} = A0_7 \times (1 - H_{L1I} - H_{L2I}) + A0_8$$

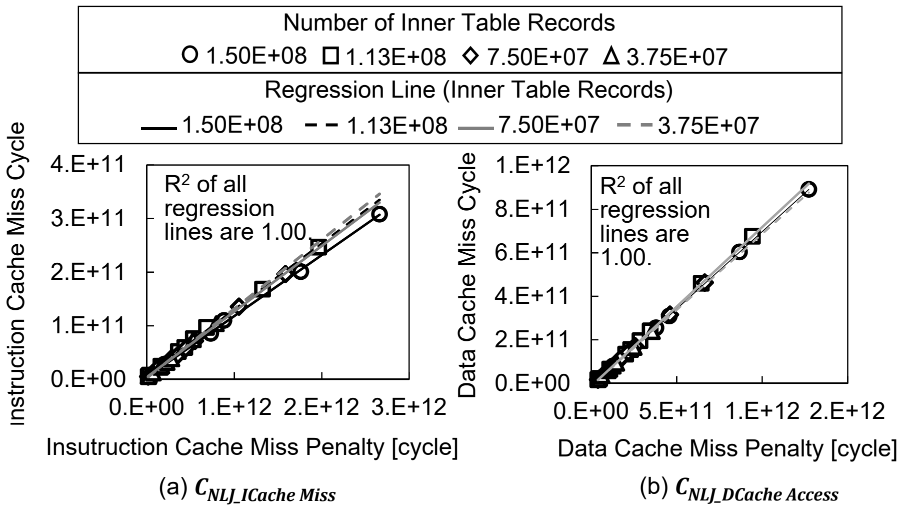


Figure 6.8.: Relation between Sum of Memory Accesses and Instruction Miss Penalty Cost or Data Cache Access Cost on NLJ

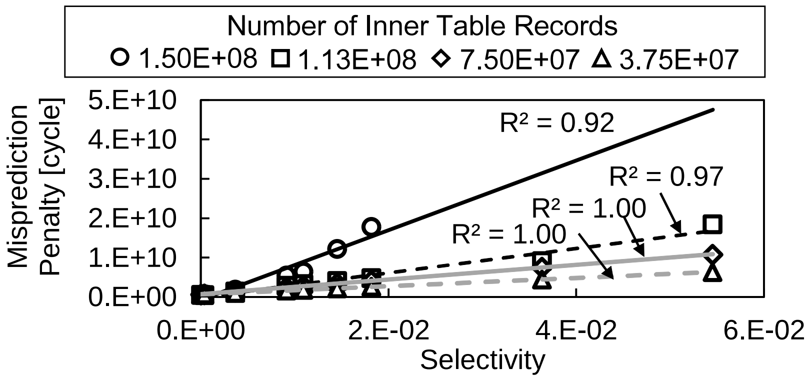


Figure 6.9.: Relation between Selectivity and Branch Misprediction Cycle on NLJ

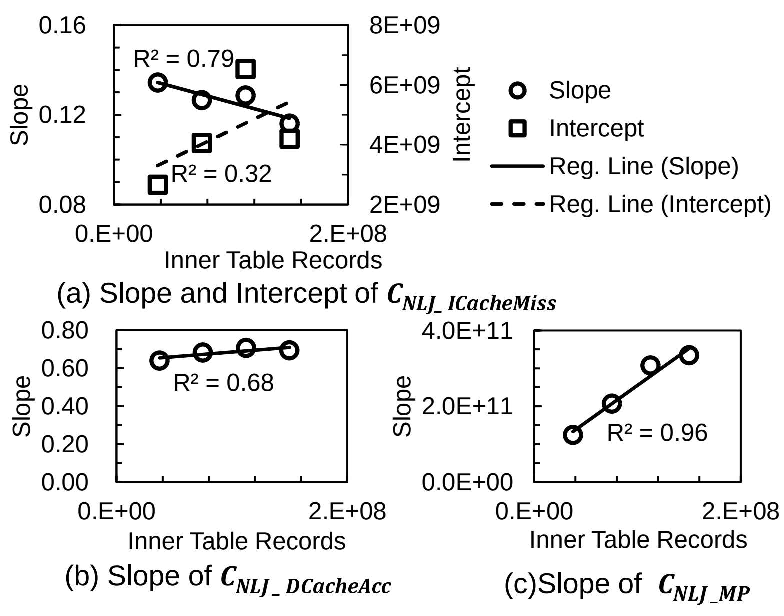


Figure 6.10.: Slope and Intercept of Regression Line of Data Access Cycles and Branch Misprediction on NLJ

$$C_{NLJ\_MP} = \{(K1_1 \times R_I + S1_1) \times P \times R_O\} \times L_{MP} \tag{6.2}$$

$$\begin{aligned} C_{NLJ\_DCacheAcc} = I_{load} \times (&H_{L1D} \times L_{L1} + H_{L2D} \times L_{L2} + H_{LLLCD} \times L_{LLLC} + H_{RLLCI} \\ &\times L_{RLLC} + H_{LMMI} \times L_{LMM} + H_{RMMI} \times L_{RMM}) \times (K2_1 \times R_I + S2_1) \end{aligned} \tag{6.3}$$

where

$I_{load} = (K2_2 \times R_I + S2_2) \times R_O \times P + (K2_3 \times R_I + S2_3)$

$H_{L1D} = e^{-A2_1} \times I_{load}^{-A2_2} + A2_3$

$H_{L2D} = A2_4 \times (1 - H_{L1D}) + (K2_4 \times R_I + S2_4)$

$H_{LLLCD} = A2_5 \times (1 - H_{L1D} - H_{L2D}) + (K2_5 \times R_I + S2_5)$

$H_{RLLCD} = A2_6 \times (1 - H_{L1D} - H_{L2D} - H_{LLLCD}) + A2_7$

$H_{LMMD} = A2_8 \times (1 - H_{L1D} - H_{L2D} - H_{LLLCD}) + (K2_6 \times R_I + S2_6)$

$H_{RMMD} = A2_{10} \times (1 - H_{L1D} - H_{L2D} - H_{LLLCD}) + A2_{11}$

6.2.2. Measurement Results of HJ Build Phase

Figure 6.11 shows the graphs of instruction access and data access in the build phase of HJ. As shown in Figure 6.11(b), the data reference has a constant value. This is because the

Table 6.3.: Parameter Setting of NLJ

Slope	Intercept	Figure Number	Focused Object
$K0_1$	$S0_1$	6.10(a)	Regression line of slope
$K0_2$	$S0_2$	6.2(b)	Regression line of slope
$K0_3$	$S0_3$	6.2(c)	Regression line of intercept
$K0_4$	$S0_4$	6.3(d)	Regression line of slope
$K0_5$	$S0_5$	6.3(d)	Regression line of intercept
$K0_6$	$S0_6$	6.3(d)	Regression line of dummy variable
$K0_7$	$S0_7$	6.5(a)	Regression line of slope
$K0_8$	$S0_8$	6.5(a)	Regression line of intercept
$K0_9$	$S0_9$	6.5(b)	Regression line of slope
$K1_1$	$S1_1$	6.10(c)	Regression line of slope
$K2_1$	$S2_1$	6.10(b)	Regression line of slope
$K2_2$	$S2_2$	6.6(a)	Regression line of slope
$K2_3$	$S2_3$	6.6(a)	Regression line of intercept
$K2_4$	$S2_4$	6.6(b)	Regression line of intercept
$K2_5$	$S2_5$	6.6(c)	Regression line of intercept
$K2_6$	$S2_6$	6.6(e)	Regression line of intercept

Parameter	Figure Number	Object
$A0_1$	6.12(a)	Average of intercept
$A0_2$	6.5(b)	Average of intercept of local LLC
$A0_3$	6.5(c)	Average of slope of remote LLC
$A0_4$	6.5(c)	Average of intercept of remote LLC
$A0_5$	6.5(d)	Average of slope of local main memory
$A0_6$	6.5(d)	Average of intercept of local main memory
$A0_7$	6.5(e)	Average of slope of remote main memory
$A0_8$	6.5(e)	Average of intercept of remote main memory
$A2_1$	6.7	Average of intercept
$A2_2$	6.7	Average of slope
$A2_3$	6.7	Average of dummy variable
$A2_4$	6.6(b)	Average of slope
$A2_5$	6.6(c)	Average of slope
$A2_6$	6.6(d)	Average of slope
$A2_7$	6.6(d)	Average of intercept
$A2_8$	6.6(e)	Average of slope
$A2_9$	6.6(e)	Average of intercept
$A2_{10}$	6.6(f)	Average of slope
$A2_{11}$	6.6(f)	Average of intercept
$d0$	6.3(d)	Dummy variable of slope
$d1$	6.3(d)	Dummy variable of intercept
$d2$	6.3(d)	Dummy variable of dm1

entire outer table is scanned even if the selectivity changes, and hence, the data access amount does not change. Considering that the number of records in the outer table is proportional to the number of data references, in this study, the intercept of the HJ cost calculation formula is expressed by a linear expression with respect to the number of records in the outer table. The instruction access increases with the increase in the selectivity because of the increase in the registration to the hash table.

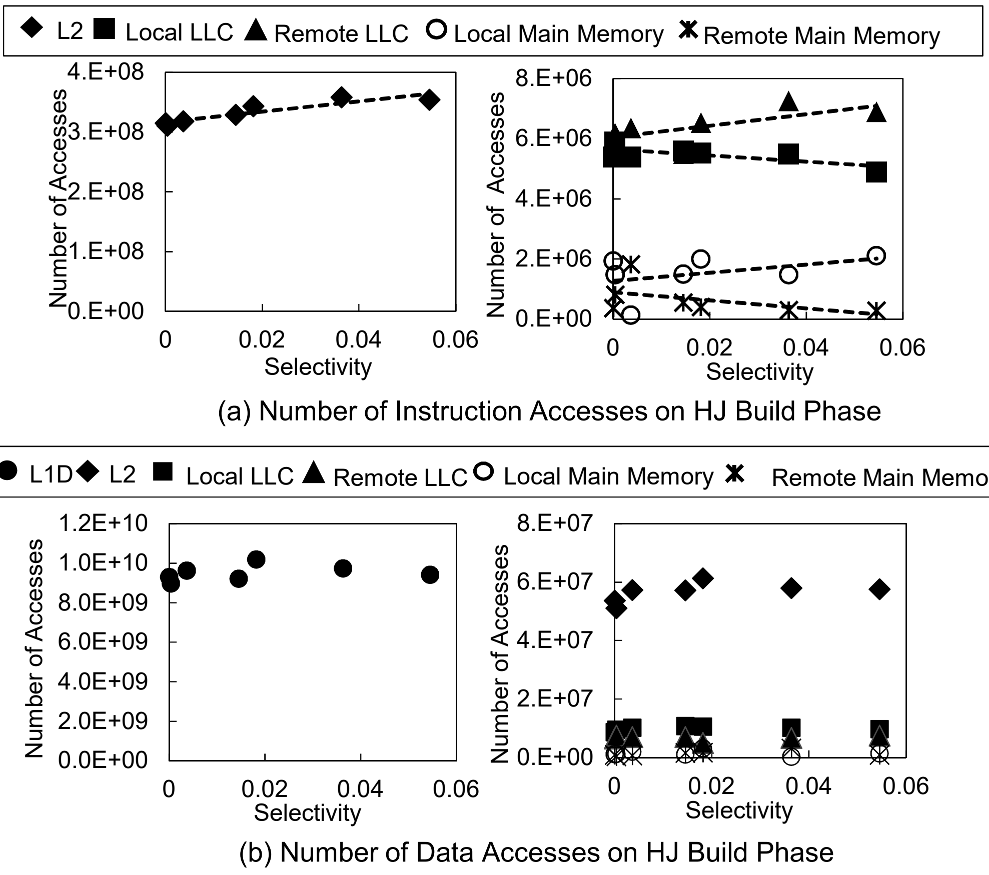


Figure 6.11.: Number of Instruction References and Data References in Build Phase of HJ

$C_{Build\_ICacheiss}$ is modeled by its average value because the coefficient of determination of linear regression analysis of the instruction miss penalty and $C_{Build\_ICacheiss}$ is small (Figure 6.12(a)).

6.2.3. Measurement Results of HJ Probe Phase

In Figure 6.13(a), the graph of the number of instruction accesses to the memory of any memory hierarchy is a straight line passing through the origin as in the NLJ. The number of data references is almost constant similar to that in the HJ build phase (Figure 6.13(b)).

As the inner table access is also a table scan, the data access in the probe phase has the same tendency as that in the build phase. Regarding instruction access, if the selectivity of the outer

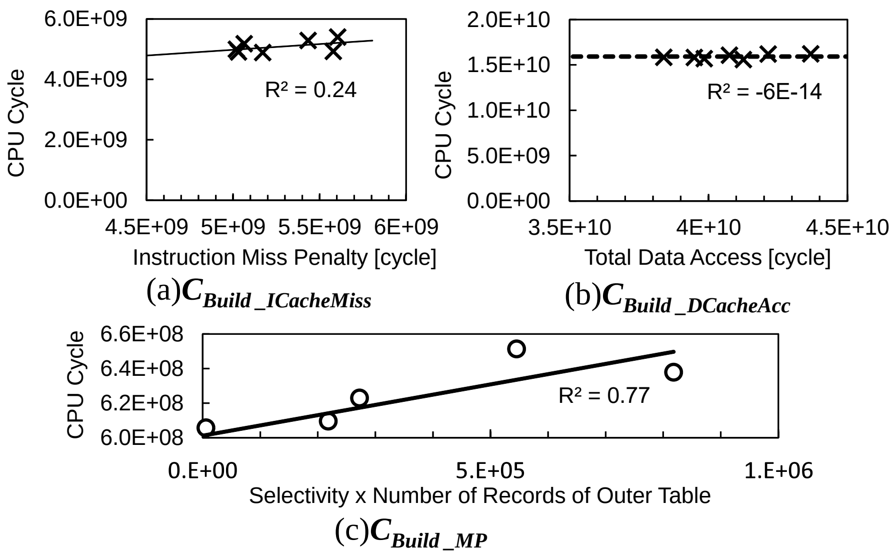


Figure 6.12.: Slope and Intercept of Regression Line of Data Access Cycles and Branch Misprediction in Build Phase of HJ

table is zero, i.e., if there is no record to be searched, the graph of instruction access can be approximated by a straight line passing through the origin.

The instruction miss penalty and $C_{Probe\_ICacheMiss}$ can be approximated by a straight line to the selectivity (Figure 6.14(a)). Similarly, for $C_{Probe\_DCacheAcc}$, the coefficient of determination is one from the result of regression analysis of the total data reference (total data access), and it can be approximated by a straight line (Figure 6.14(b)). Similarly, C_{Probe_MP} can be approximated by a straight line to the selectivity as shown in Figure 6.14(c).

Based on the above considerations, the cost formula for HJ is expressed in equations (6.4), (6.5), (6.7), (6.8), and (6.9). The number of inner tables when measuring CPU events is given by R_{O_ref}. Table 6.4 shows the definitions of variables.

$$C_{Build\_ICacheMiss} = A3_7 \times (R_O / R_{O\_ref}) \tag{6.4}$$

$$\begin{aligned} C_{Probe\_ICacheMiss} = [K3_1 \times \{ & (K3_2 \times P + S3_2) \times L_{L2} \\ & + (K3_3 \times P + S3_3) \times L_{LLLC} + (K3_4 \times P + S3_4) \times L_{RLLC} \\ & + (K3_5 \times P + S3_5) \times L_{LMM} + (K3_6 \times P + S3_6) \times L_{RMM} \} \\ & + S3_1] \times (R_O / R_{O\_ref}) \end{aligned} \tag{6.5}$$

$$C_{Build\_MP} = K3_9 \times P \times R_O + S3_9 \tag{6.6}$$

$$C_{Probe\_MP} = K3_7 \times P \times R_O + S3_7 \tag{6.7}$$

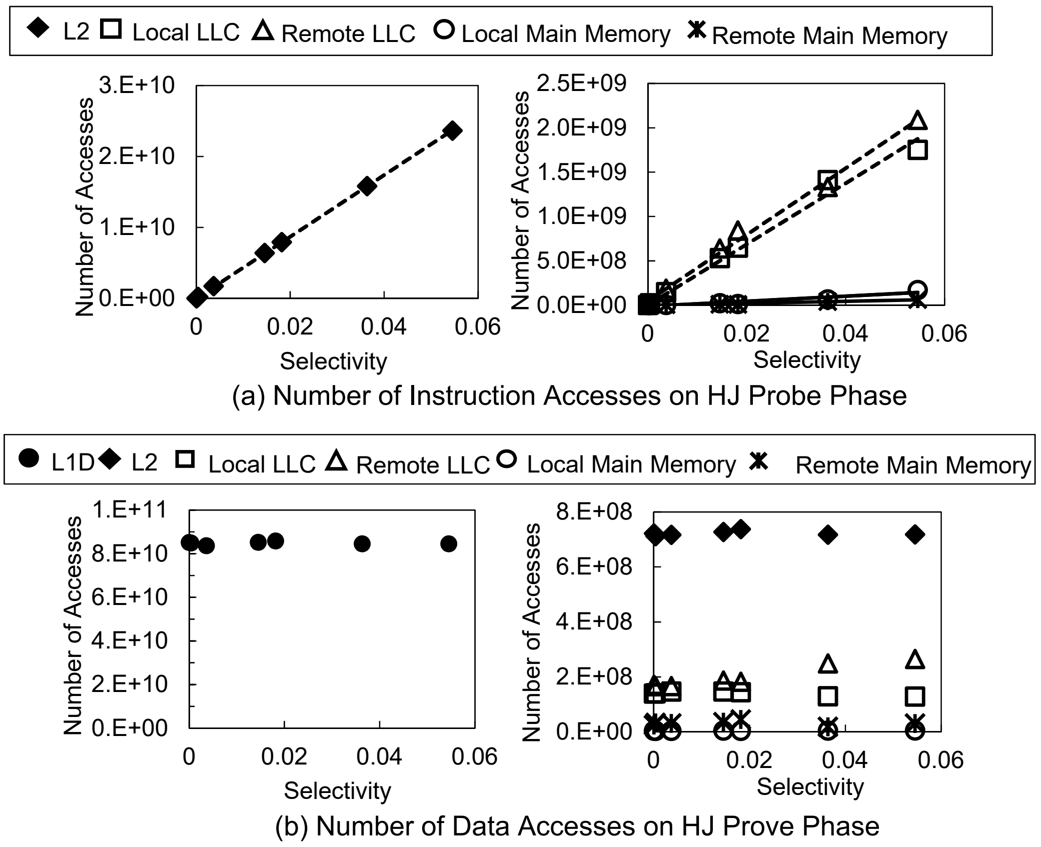
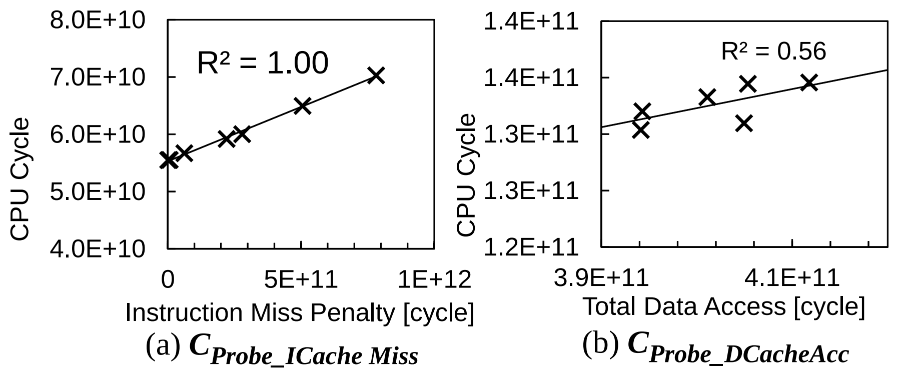
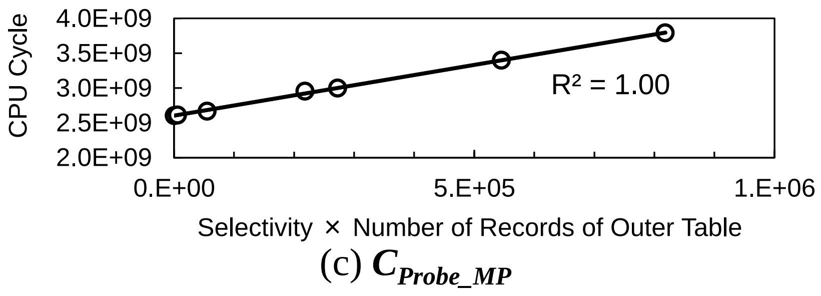


Figure 6.13.: Number of Instruction References and Data References in Probe Phase of HJ



(a) $C_{Probe\_ICache\ Miss}$ (b) $C_{Probe\_DCacheAcc}$



(c) C_{Probe_MP}

Figure 6.14.: Slope and Intercept of Regression Line of Data Access Cycles and Branch Misprediction in Probe Phase of HJ

$$C_{Build\_DCacheAcc} = A3_8 \times (R_RO/R_{O\_ref}) \tag{6.8}$$

$$\begin{aligned} C_{Probe\_DCacheAcc} = \{&K3_8 \times (A3_1 \times L_{L1} + A3_2 \times L_{L2} \\ &+ A3_3 \times L_{LLLC} + A3_4 \times L_{RLLC} \\ &+ A3_5 \times L_{LMM} + A3_6 \times L_{RMM}) + S3_8\} \times (R_O/R_{O\_ref}) \end{aligned} \tag{6.9}$$

Table 6.4.: Parameter Setting of HJ

Slope	Intercept	Figure Number	Focused Object
$K3_1$	$S3_1$	6.12(a)	Regression line
$K3_2$	$S3_2$	6.13(a)	Regression line of L2
$K3_3$	$S3_3$	6.13(a)	Regression line of local LLC
$K3_4$	$S3_4$	6.13(a)	Regression line of remote LLC
$K3_5$	$S3_5$	6.13(a)	Regression line of local main memory
$K3_6$	$S3_6$	6.13(a)	Regression line of remote main memory
$K3_7$	$S3_7$	6.14(c)	Regression line of branch misprediction
$K3_8$	$S3_8$	6.14(b)	Regression line
$K3_9$	$S3_9$	6.12(c)	Regression line
	Other	Figure Number	Focused Object
	$A3_1$	6.13(b)	Average of L1D
	$A3_2$	6.13(b)	Average of L2
	$A3_3$	6.13(b)	Average of local LLC
	$A3_4$	6.13(b)	Average of remote LLC
	$A3_5$	6.13(b)	Average of local main memory
	$A3_6$	6.13(b)	Average of remote main memory
	$A3_7$	6.12(a)	Average
	$A3_8$	6.12(b)	Average

6.3. Evaluation of Cost Calculation Results

For the three combinations of the Customer table and Order table, Supplier table and Lineitem table, and Part table and Lineitem table, which are combinations of two tables in which relations are set and whose capacity is large, from the table of TPC-H evaluate accuracy, the measured values used for parameter setting of the cost calculation formula are the actual values measured when joining the Customer table and the Order table and for the remaining two combinations, the cost calculation is performed using only the number of rows and selectivity of the table. To compare the execution time of the query, the actual I/O processing time is added to the CPU cost calculated using the proposed method as the predicted value of the proposed method. The memory latency refers to the literature [52].

The cost (“proposed” in Figure 6.15) calculated using the above method and the actual query processing time of the DBMS (including both CPU and I/O) were compared (Figure 6.15 (a), (c), (e)). Furthermore, the actual query processing time and the costs which is described as “conventional” in Figure 6.15 and obtained from the cost calculation formula of the open source DBMS in Equation (2.1), (2.2), and (2.3) are also compared. However, as HJ is not supported in the existing method, it is considered as a sum of single-table scans of the outer and inner tables. The task set in this study is to determine the cross point of NLJ and HJ graphs accurately in determining the join method. In the cases evaluated here, the accuracy improvement ratio *AIR* is given by the following equation (6.10).

$$AIR = \frac{|S_{proposed} - S_{measured}|}{|S_{conventional} - S_{measured}|} \tag{6.10}$$

where $S_{proposed}$ is selectivity of the cross point obtained by the proposed cost calculation method, $S_{measured}$ is selectivity of the measured cross point, and $S_{conventional}$ is selectivity of the cross point obtained by the conventional cost calculation method.

The accuracy improvement ratio *AIR* is an index showing how close the cross point obtained by the proposed cost calculation method is to the measured cross point in comparison with the cross point by the conventional method. It is found that the difference between the predicted cross point and the measured cross point was less than 10^{-3} (0.1%) selectivity, and the proposed method could estimate the cross point with 83% to 94% accuracy improvement ratio *AIR* in Table 6.5. Improvement of prediction accuracy was achieved compared with accuracy of the conventional method.

Table 6.5.: Difference of Cross Point and Improvement Ratio

Join tables	C-O	P-L	S-L
Conventional method	1.5×10^{-2}	3.5×10^{-3}	2.2×10^{-3}
Proposed method	5.0×10^{-4}	6.0×10^{-4}	1.4×10^{-4}
Improvement ratio *AIR*	97%	83%	94%

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

In the conventional model mentioned above, it is assumed that the unit cost of CPU is $CPR = 0.2$ and that of I/O is $CPIO = 1$, which are the default defined values. However,

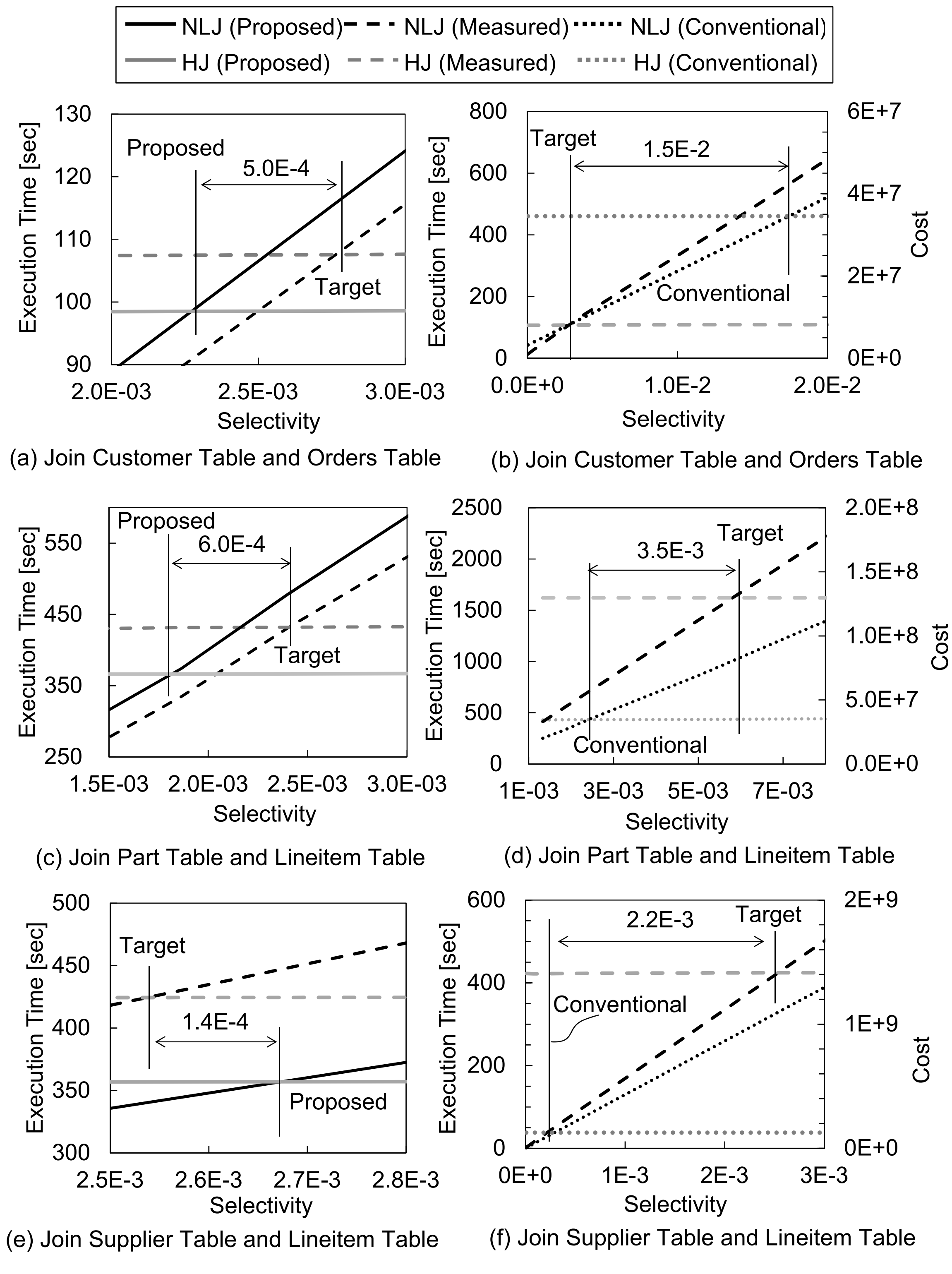


Figure 6.15.: Comparison of Measured Results, Proposed Cost Model and Conventional Cost Model

considering the CPU unit cost means CPU cycle time or memory access latency and the I/O unit cost means I/O access latency, it is not reasonable that the unit cost of CPU is 0.2 and that of I/O is 1 because DB administrators can tune parameters about cost calculation for optimizing queries. Therefore, we evaluated the case in which the unit cost of CPU is the ratio of CPU cycle time and measured I/O latency ($CPR = 3 \times 10^{-6} = (1/2.13GHz)/154ms$) and the case in which the unit cost of CPU is the ratio of main memory latency and measured I/O latency ($CPR = 6 \times 10^{-4} = 100ns/154ms$). These conventional model are called *t*he updated conventional model below. The results of joining two tables, customer and orders (C-O), supplier and lineitem (S-L), part and lineitem (P-L) are shown in the Figure 6.16(a)(b)(c).

Table 6.6 shows that the difference between estimating cross point using the unit cost considering CPU cycle time or main memory latency measured cross point are smaller than that of the default CPU unit cost case. However, the difference between the cross point estimated by proposed method and the measured cross point is smaller than updated conventional method with 71% to 94% accuracy improvement ratio. Therefore, our proposed cost calculation method can be estimated cross point more accurately than the default method.

Table 6.6.: Difference of Cross Point and Improvement Ratio with Updated Conventional Method

Join tables	C-O	P-L	S-L
CPU cost parameter type	MEM	MEM	MEM
Updated conventional method	1.8×10^{-3}	2.1×10^{-3}	2.5×10^{-3}
Proposed method	5.0×10^{-4}	6.0×10^{-4}	1.4×10^{-4}
Improvement ratio *AIR*	72%	71%	94%

Note1: C: Customer, O: Orders, L: Lineitem, P: Part, S: Supplier
Note2: MEM is *main memory latency*.

6.4. Discussion

In this study, We proposed the cost calculation method based on CPU statistical information for optimizing database queries and evaluated its effectiveness using a large database. It is found that our proposed cost calculation method can estimate the cross point closer than the conventional method to the measured value. As the evaluation environment, the TPC-H benchmark database is used for the evaluation of the proposed method. TPC-H has an advantage in that it is easier to analyze the evaluation result because the distribution of data is uniform. However, actual data have a bias in the distribution of attribute values. In the cost calculation formula obtained in this study, the cost is determined only by the selectivity, and the same measurement result can be obtained if the selectivity is the same, regardless of the distribution of the data. The accuracy of the cost calculation formula in this study depends on the accuracy of the selectivity. As a general DBMS acquires attribute values and their distribution information in a database in the form of a histogram at the time of loading data, the proposed method can be applied to an actual DBMS easily.

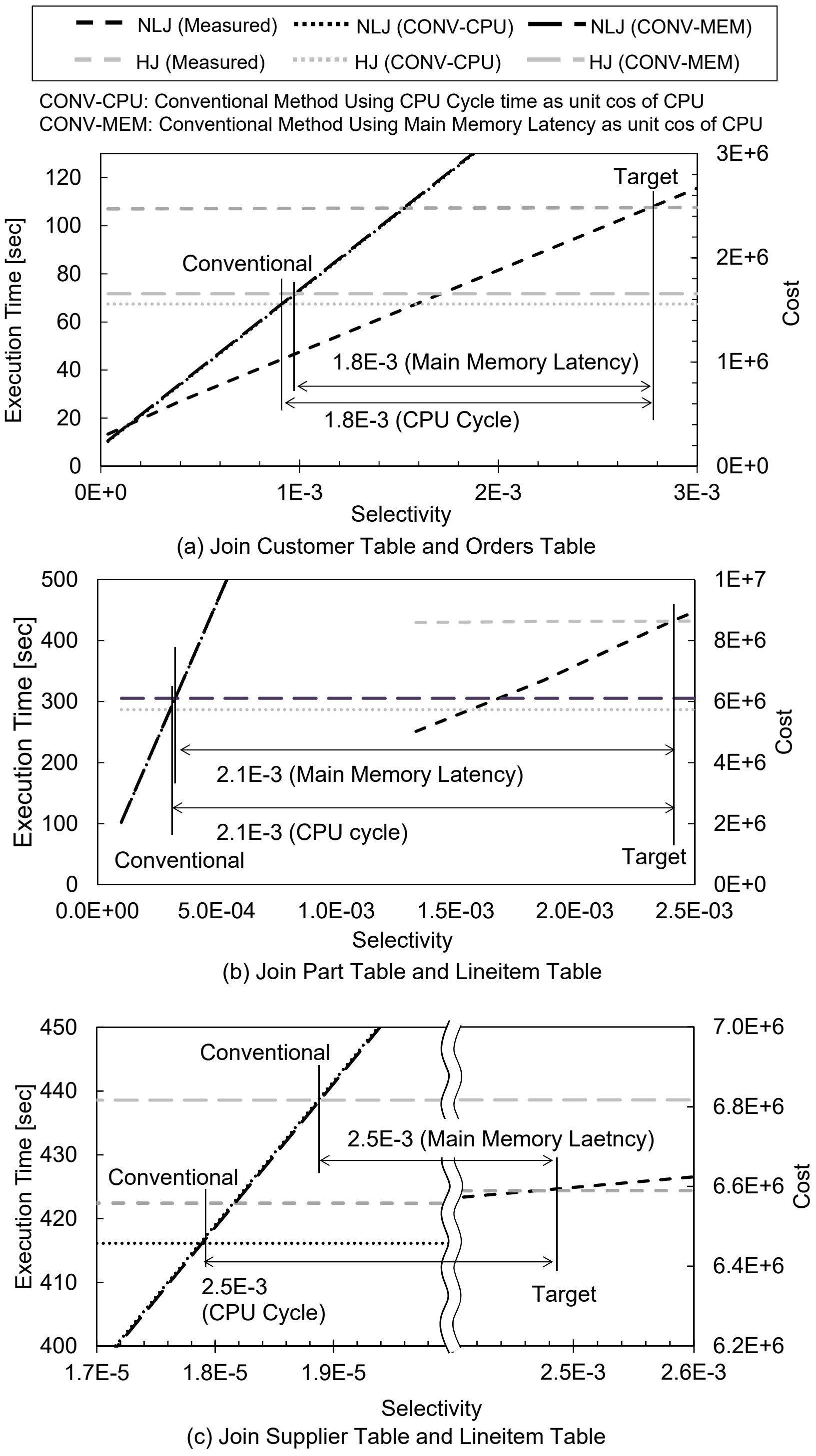


Figure 6.16.: Comparison of Measured Results, Proposed Cost Model and Conventional Cost Model Considering CPU Cycle Time or Main Memory Latency

In addition, as the present technique sets parameters of the cost calculation formula based on measurement, it is difficult to deal with various patterns such as the presence or absence of indexes and complicated queries. Although we focused on the operation of all CPUs here, it is necessary to improve accuracy using a model with fewer parameters for a practical application.

6.5. Conclusion of Feasibility Study

For database query optimization, we proposed a cost calculation method focusing on CPU architecture and presented the proposal and evaluation of a cost calculation formula reflecting the actual measurement result. In the cost calculation method, CPU processing time is classified into three types based on the characteristics of instruction processing. Subsequently, CPU cost calculation formulas using actual measurement values are obtained. In the evaluation experiment, the difference between the selectivity at which the join method is switched based on the obtained cost formula and the selectivity at actual measurement is less than 0.1% in the selectivity of the data. Consequently, the proposed cost calculation method can calculate the cost with high accuracy. The appropriate join method can be selected by applying the proposed method in this study, and the possibility of reducing the risk of unexpected query execution delay to users of DBMS was obtained.

7. Measurement-based Cost Estimation Method for Multi-Table Join Operation

In chapter 6, it was found that the proposed measurement-based cost calculation method can improve accuracy of cost in the case of two table join. In this chapter, the proposed method is generalized so that it can be applied to multiple-table joins. In addition, the database size is required to be smaller than that of Chapter 6 to shorten the measurement time of CPU statistical information for minimizing database administrator's operation cost. Therefore, we extend the measurement-based cost calculation method to support multi-table join using measured data on small database.

Moreover, in query optimization, not only the join method but also the order of the tables to be joined is determined. In this study, the join operation is performed by the left-deep join tree [53] because MariaDB uses the left-deep join tree for query optimization. The left-deep join tree has the feature that the amount of memory usage is small, and the join operation is completed in one-pass. However, when executing small size of query, performance of bushy tree is better than liner tree such as left-deep and right-deep tree, and when executing query parallel, performance of right-deep-tree is better than left-deep-tree [54]. Therefore, it is one of our future work whether or not the proposed method can be widely applied in various join trees.

In general, in query optimization, combinations of tables and the order of joining the tables are created under the condition of having the same attributes to each table, and their costs are calculated. Then, the combination with the smallest cost among them is selected. In this study, we compare the actual processing time in the case of joining two or more tables with the cost calculated using our proposed cost calculation method. We verify that our proposed cost calculation method can optimize join operation thereby finding the minimum query execution time case.

7.1. Proposal and Verification of Measurement-based Cost Estimation Method

7.1.1. Measurement of Parameters for Creation of Cost Formula

To obtain the parameters in Table 5.1, actual measurements were made. The measurement environment is listed in Table 7.1. 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 [51] 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 7.1.: Parameter Measurement Environment

CPU	Xeon L5630 2.13 GHz 4-core, LLC 12 MB [Westmere-EP]) ×2
Memory	DDR3 12 GB (4 GB ×3) physically attached to only one CPU
Disk (DB)	PCIe NVMe Flash SSD 800 GB ×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 7.1. In the SQL statement, we modified Query 3 of TPC-H for an evaluation of two-table join and extracted only join processing (Figure 7.1(a)). The order of joining tables is shown in Figure 7.1(b). This query access path 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 which are defined in the specification of TPC-H [5].

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 7.1(c)). As for NLJ, the selectivity and number of records of the inner table were changed (Figure 7.1(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 7.1(e)). In the probe phase, only the inner table was accessed, and the number of records of the inner table was changed (Figure 7.1(d)). In order to accurately measure the CPU events of the build phase, the empty inner table (Figure 7.1(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:

$$\begin{aligned}&(\textit{Number of CPU events during probe phase})\\&\quad=(\textit{Number of CPU events during total query execution})\\&\qquad-(\textit{Number of CPU events during build phase})\end{aligned}\tag{7.1}$$

For the combination case, the query and its measurement conditions are shown in Figure 7.2. This query is based on Query 3 of TPC-H. The order of joining tables is shown in Figure 7.2(b). This query access path is generated by MariaDB. The search condition and selectivity of the outer table are shown in Figure 7.2(c). This condition is used for modeling Figure 4.2(c-1). In addition, the search condition and those of the inner tables are shown in Figure 7.2(d). This condition for the inner table1 and the inner table2 was used for modeling Figure 4.2(c-2). The search condition (e) in Figure 7.2 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) [44] 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 A.1, A.2 and A.3 in Appendix A.2.

It is necessary to analyze not only CPU time but also I/O operation time to estimate the whole execution time of a query as shown in Equation 2.1. 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.

7.1.2. Measurement Results and Cost Calculation Formulas of Join Operation

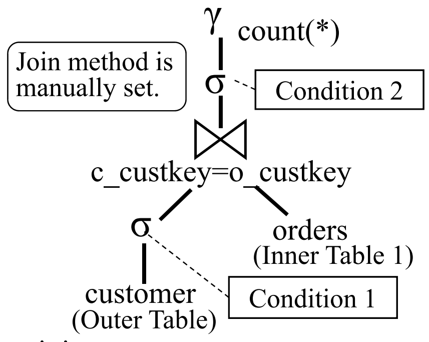
In this study, we investigate the relationships between selectivity, number of instructions, number of events related to memory reference, and number of branch mispredictions. For NLJ, the number of instructions and number of memory references are expected to increase because the number of records accessed 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.

(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) Access Path



(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^{-0}

(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

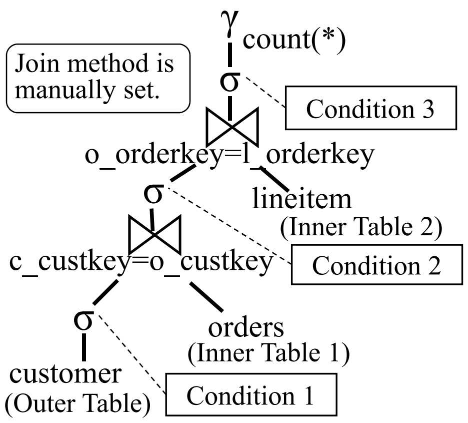
Number of Records in Inner Table	7,500,000	5,625,000	3,750,000	1,875,000
Number of Records in Outer Table	0			

Figure 7.1.: 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) Access Path



(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			

Figure 7.2.: Target Query of Measurement and Cost Estimation for Three-table Join

In addition, the relationship between the sum of the product of the references to each memory and its latency, and $C_{ICacheMiss}$ (3.7) and $C_{DCacheAcc}$ (3.10) were modeled. Here, C_{MP} (3.8) 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 [44, 55]. 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 ($\mathrm{X}_{\mathrm{cross}}$ in Figure 1.2) with the conventional method.

Figure 7.3(1) shows the relationship between the number of records the DBMS accessed and load instructions. Figure 7.3(7) shows the relationship between the total number of accessed records and number of instructions. The number of accessed 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 7.2. 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 7.3(2) and (8) show the relationship between the number of instructions executed by the DBMS and the number of L1 cache hits. Figures 7.3(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 7.2. 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 7.3(13) shows the relationship between the number of records accessed for the join operation and the branch misprediction cycles C_{MP}. Figure 7.3(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 7.3(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 7.2.

Figures 7.4(1)–(15), Figure 7.5(1)–(15) and Figures 7.6(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 7.2 and Table 7.3. 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 are used for creating the cost calculation model.

In particular, the slope of the regression line in Figures 7.3(2)–(5) and (9)–(11); Figures 7.4(2)–(5), (9)–(11), Figures 7.5(2)–(5), and (9)–(11); and Figures 7.6(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.

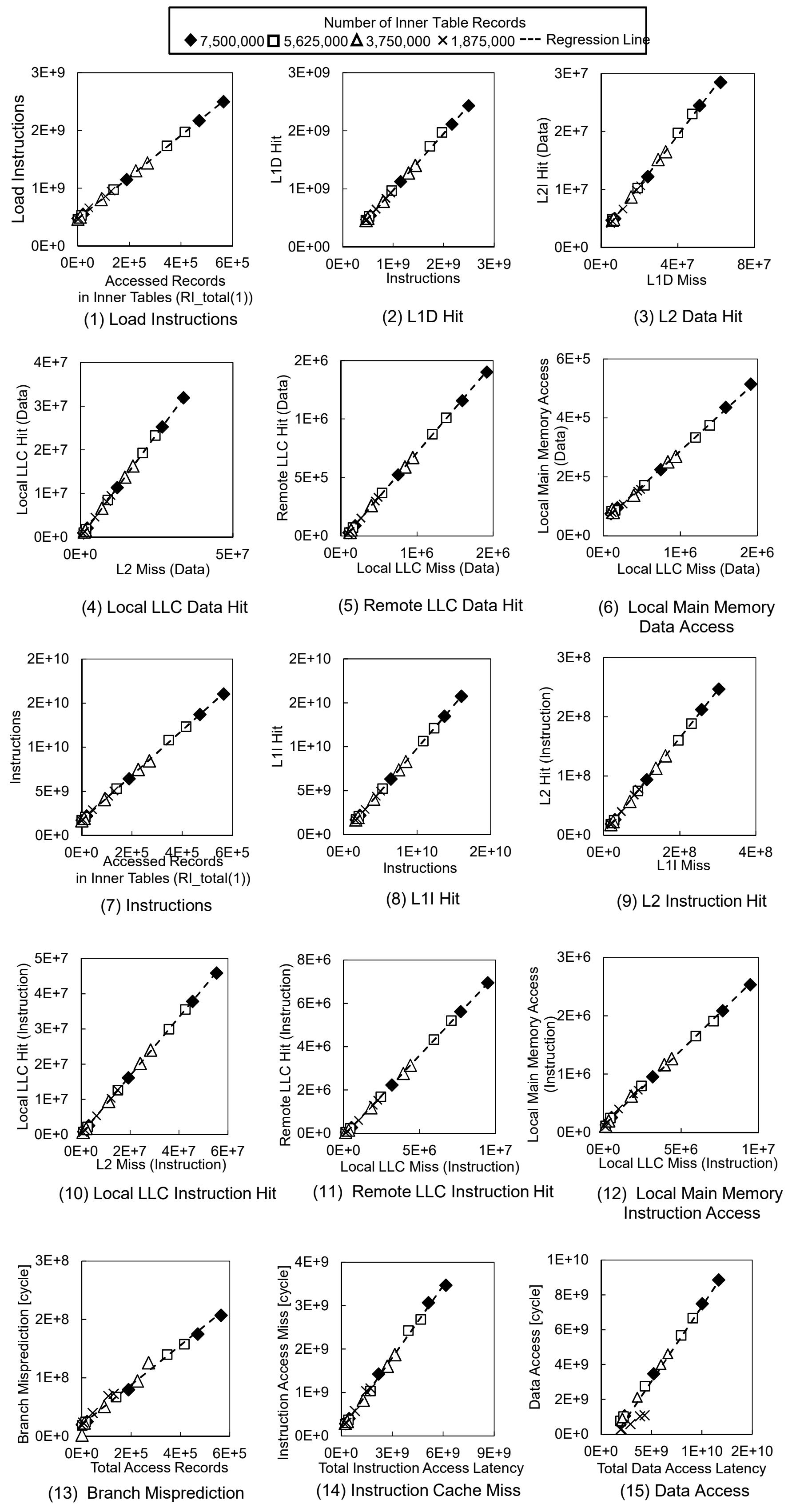


Figure 7.3.: CPU Event Count on Executing NLJ

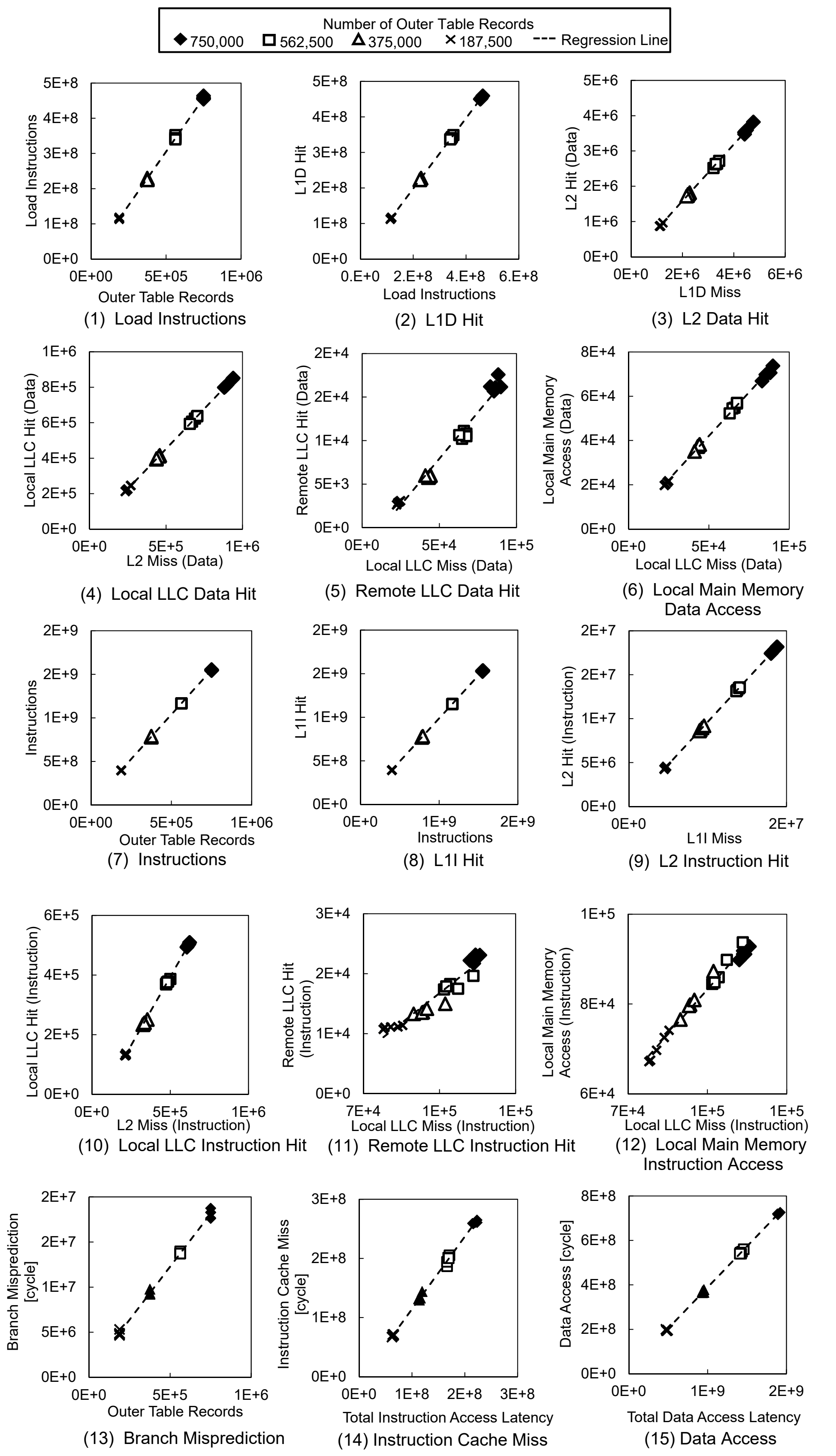


Figure 7.4.: CPU Event Count on Executing Build Phase of HJ

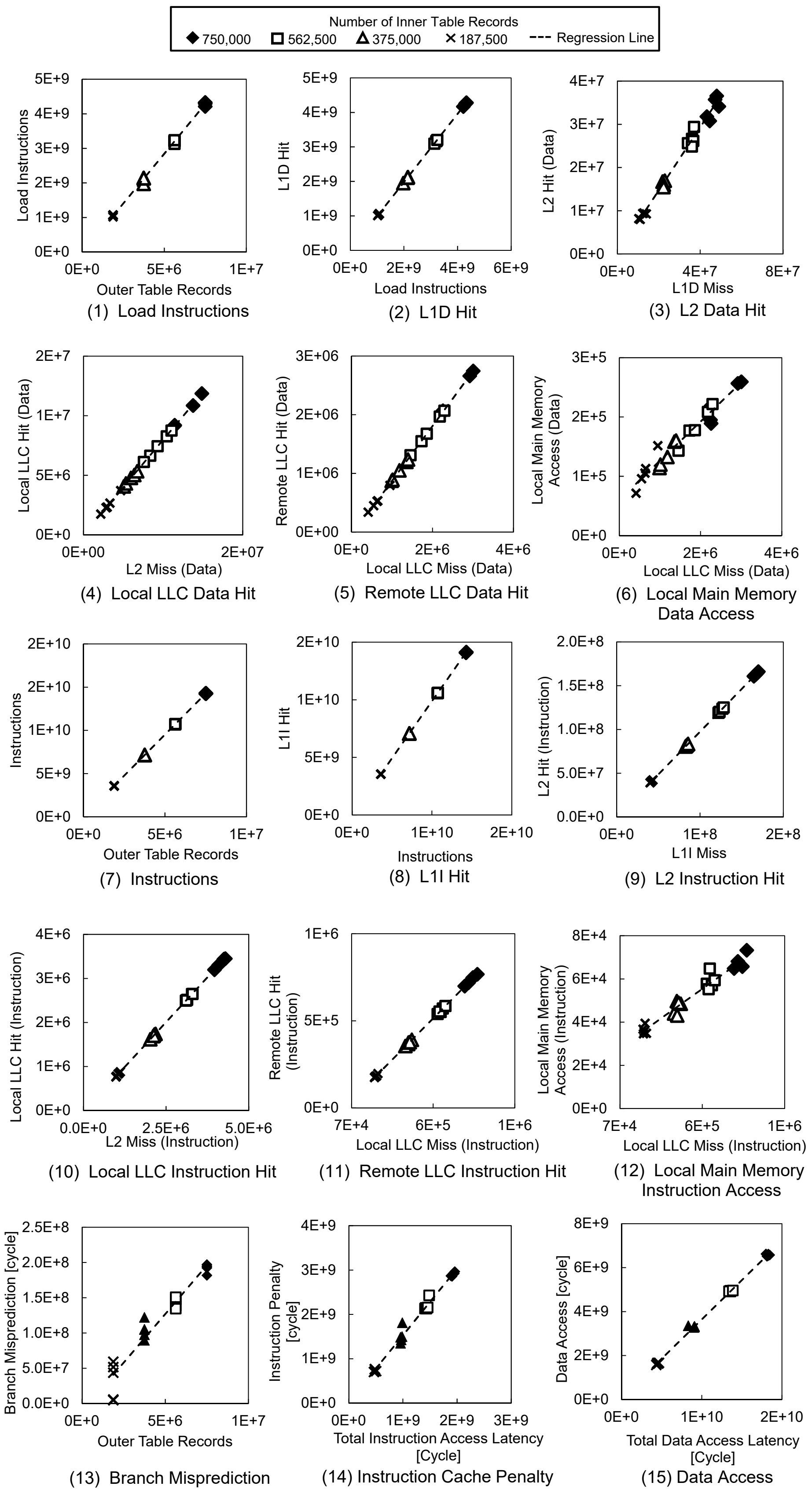


Figure 7.5.: CPU Event Count on Executing Probe Phase of HJ

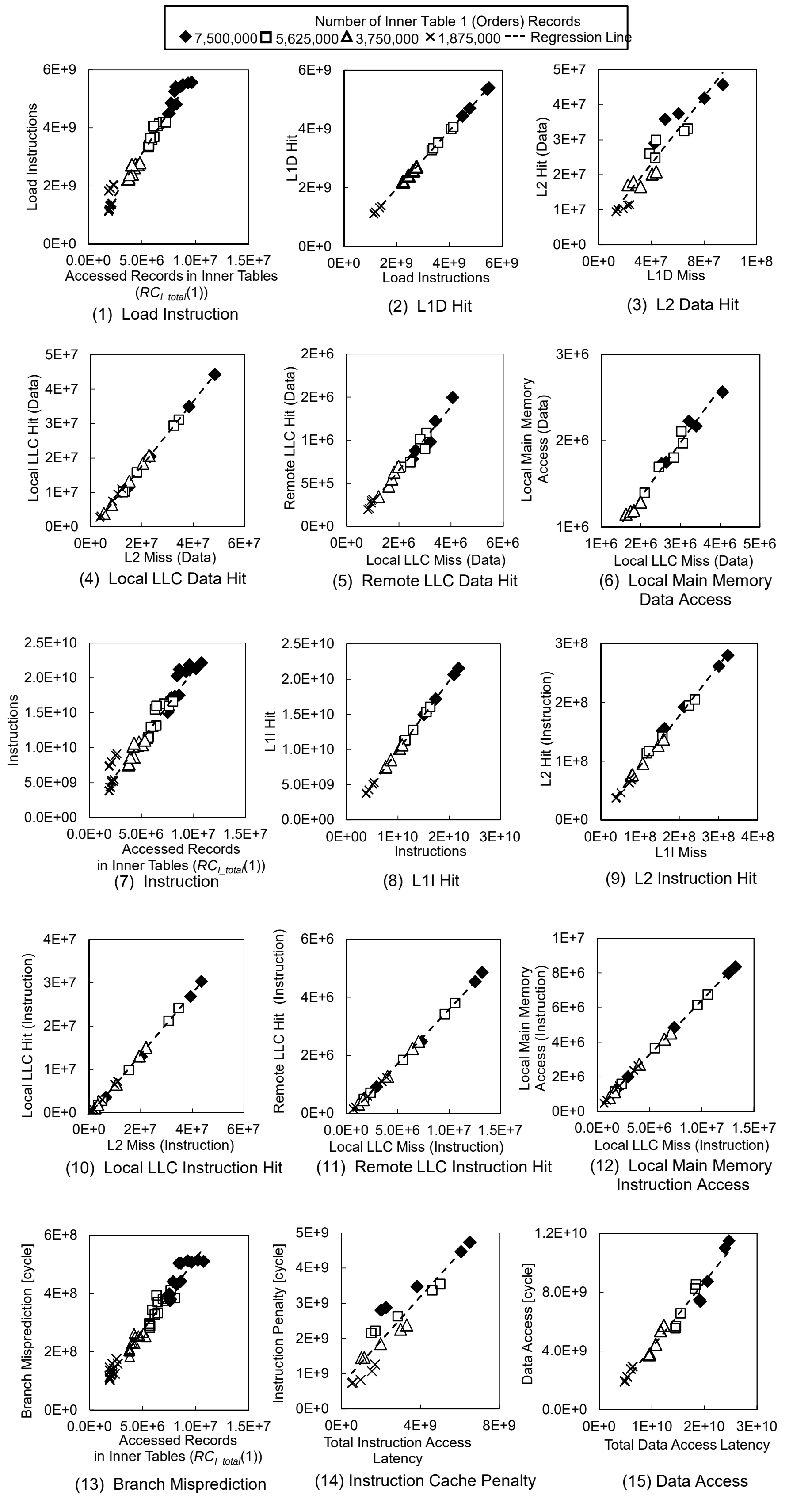


Figure 7.6.: CPU Event Count on Combination Probe Phase

In this study, the number of cache hits is chosen as an explanatory variable, as shown in Figure 7.7(a), which is the same graph as that in Figure 7.3(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 7.7(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 (7.2) and (7.3).

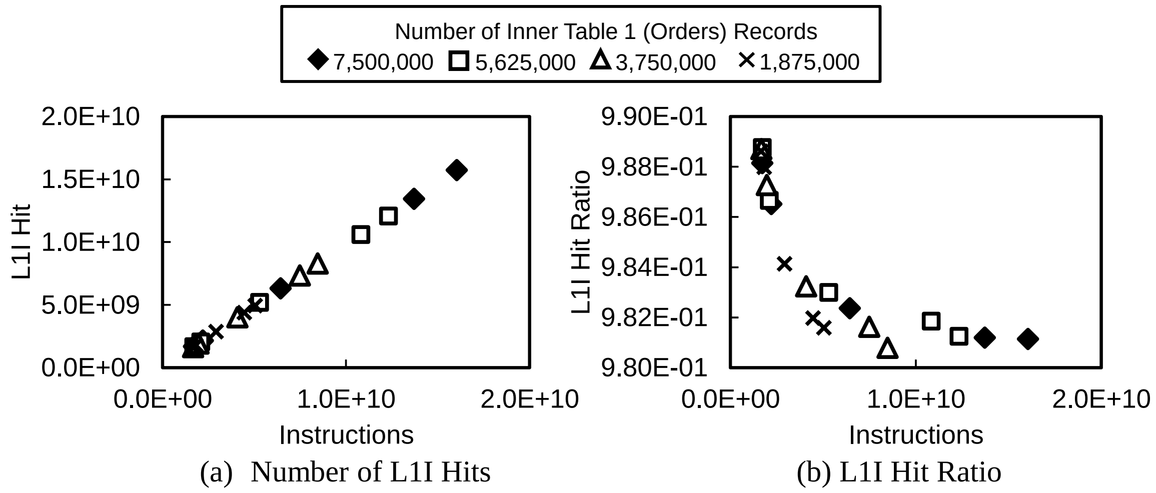


Figure 7.7.: Problem of Modeling Cache Hit Ratio

The regression line of Figure 7.7(a) is

$$M_{L1I} = A \times I + B, \tag{7.2}$$

where A and B are the slope and intercept of a linear regression on the two table join in Figure 7.1, respectively. The cache hit ratio is obtained by dividing the number of cache hits by that of instructions. Therefore, the cache hit ratio (7.3) is obtained by dividing both sides of (7.2) by I. The equation 7.3 is a hyperbolic.

$$(L1I\ Hit\ Ratio) = A + \frac{B}{I}, \tag{7.3}$$

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 4.1(a), Figure 4.2 (c-2)). As shown in Figure 7.8, 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} (7.4). 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 (7.4) and (7.5).

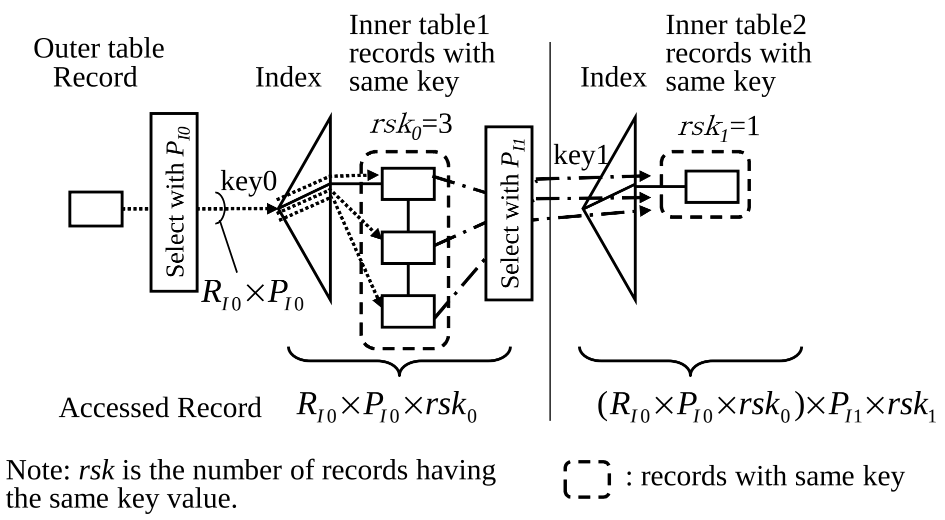

Outer table Record
Index
Inner table1 records with same key
Index
Inner table2 records with same key
Select with P_{I0}
key0
$R_{I0} \times P_{I0}$
rsk_0=3
Select with P_{I1}
key1
rsk_1=1
Accessed Record
$R_{I0} \times P_{I0} \times rsk_0$
$(R_{I0} \times P_{I0} \times rsk_0) \times P_{I1} \times rsk_1$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

Figure 7.8.: Simple Example of Number of Records Having the Same Key Value (rsk)

$$rsk_i = \begin{cases} \dfrac{R_{Ii}}{R_{I(i-1)}} & R_{Ii} \geq \max\{R_{I(j)},\ j = 0, 1, \cdots, i-1\} \\ 1 & R_{Ii} < \max\{R_{I(j)},\ j = 0, 1, \cdots, i-1\} \end{cases} \quad \text{where} \quad R_{I0} = R_O \tag{7.4}$$

$$R_{I\_total(n)} = R_O \times \sum_{j=1}^{n} \prod_{i=0}^{j-1} (rsk_i \times P_{Ii}) \tag{7.5}$$

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. 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 (7.6) as follows:

$$RC_{I\_total(n)} = R_O \times \left\{ \sum_{j=1}^{n} \prod_{i=0}^{j-1} (rsk_i \times P_{Ii}) \times \left(log_t(R_{Ij}) + \frac{rsk_j + 1}{2} \right) + 1 \right\} \tag{7.6}$$

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. $(rsk_j+1)/2$ is the expected value of the number of traversing pointers from each leaf node of the 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 7.3(1)(7)(13) and Figure 7.6(1)(7)(13)).

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

$$I = A1 \times R + B1 \tag{7.7}$$
$$M_{L1I} = A2 \times I + B2 \tag{7.8}$$
$$M_{L2I} = A3 \times (I - M_{L1I}) + B3 \tag{7.9}$$
$$M_{LLLCI} = A4 \times (I - M_{L1I} - M_{L2I}) + B4 \tag{7.10}$$
$$M_{RLLCI} = A5 \times (I - M_{L1I} - M_{L2I} - M_{LLLCI}) + B5 \tag{7.11}$$
$$M_{LMMI} = A6 \times (I - M_{L1I} - M_{L2I} - M_{LLLCI}) + B6 \tag{7.12}$$
$$I_{Load} = A7 \times R + B7 \tag{7.13}$$
$$M_{L1D} = A8 \times I_{Load} + B8 \tag{7.14}$$
$$M_{L2D} = A9 \times (I - M_{L1D}) + B9 \tag{7.15}$$
$$M_{LLLCD} = A10 \times (I - M_{L1D} - M_{L2D}) + B10 \tag{7.16}$$
$$M_{RLLCD} = A11 \times (I - M_{L1D} - M_{L2D} - M_{LLLCD}) + B11 \tag{7.17}$$
$$M_{LMMD} = A12 \times (I - M_{L1D} - M_{L2D} - M_{LLLCD}) + B12 \tag{7.18}$$
$$\begin{aligned} C_{ICacheMiss} = {} & A13 \times (M_{L2I} \times L_{L2} + M_{LLLCI} \times L_{LLLC} \\ & + M_{RLLCI} \times L_{RLLC} + M_{LMMI} \times L_{LMM}) + B13 \end{aligned} \tag{7.19}$$
$$\begin{aligned} C_{DCacheAcc} = {} & A14 \times (M_{L1D} \times L_{L1} + M_{L2D} \times L_{L2} \\ & + M_{LLLCD} \times L_{LLLC} + M_{RLLCD} \times L_{RLLC} \\ & + M_{LMMD} \times L_{LMM}) + B14 \end{aligned} \tag{7.20}$$
$$C_{MP} = A15 \times R + B15 \tag{7.21}$$

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 ((3.11), (7.7)–(7.21)) can become the following single formula by focusing on R ((7.22), (7.23), (7.24)). This formula suggests that our approach means estimating an accurate unit CPU cost per accessed record.

$$C_{Total} = \alpha \times R + \beta \tag{7.22}$$

where

$$\begin{aligned}
\alpha = {} & A1\times A13\times(L_{L2}\times A3\times(1-A2)\\
& +L_{LLLC}\times A4\times(1-A3-A2+A2\times A3)\\
& +L_{RLLC}\times A5\times(1-A2-A3+A2\times A3-A4+A3\times A4\\
& +A2\times A4-A2\times A3\times A4)\\
& +L_{LMM}\times A6\times(1-A2-A3+A2\times A3-A4+A3\times A4\\
& +A2\times A4-A2\times A3\times A4))\\
& +A7\times A14\times(L_{L1}\times A8+A14\times L_{L2}\times(A9-A8\times A9)\\
& +L_{LLLC}\times A10\times(1-A9-A8+A8\times A9)\\
& +L_{RLLC}\times A11\times(1-A8-A9+A8\times A9-A10+A9\times A10\\
& +A8\times A10-A8\times A9\times A10)\\
& +L_{LMM}\times A12\times(1-A8-A9+A8\times A9-A10+A9\times A10\\
& +A8\times A10-A8\times A9\times A10))+A15
\end{aligned} \tag{7.23}$$

$$\begin{aligned}
\beta = {} & A13\times(L_{L2}\times(-A3\times B2+B3)\\
& +L_{LLLC}\times(-A4\times B2+A3\times A4\times B2-A4\times B3+B4)\\
& +L_{RLLC}\times(-A5\times B2+A3\times A5\times B2-A5\times B3\\
& +A4\times A5\times B2-A3\times A4\times A5\times B2+A4\times A5\times B3\\
& -A5\times B4+B5)\\
& +L_{LMM}\times(-A6\times B2+A3\times A6\times B2-A6\times B3+A4\\
& \times A6\times B2-A3\times A4\times A6\times B2+A4\times A6\times B3-A6\times B4\\
& +B5))+A14\times(L_{L1}\times B8+L_{L2}\times(-A9\times B8+B9)\\
& +L_{LLLC}\times(-A10\times B8+A9\times A10\times B8-A10\times B9+B10)\\
& +L_{RLLC}\times(-A11\times B8+A9\times A11\times B8-A11\times B9\\
& +A10\times A11\times B8-A9\times A10\times A11\times B8+A10\times A11\times B9\\
& -A11\times B10+B11)\\
& +L_{LMM}\times(-A12\times B8+A9\times A12\times B8-A12\times B9\\
& +A10\times A12\times B8-A9\times A10\times A12\times B8+A10\times A12\times B9\\
& -A12\times B10+B11))+B13+B14+B15
\end{aligned} \tag{7.24}$$

Table 7.2 lists the definitions of the parameters given in (7.7)–(7.21) for NLJ and HJ. the calculation formula of the number of disk I/Os was created using the regression line shown in Figure 7.9(a).

Since all the records of outer table are accessed in NLJ, if the size of outer table is larger than the buffer cache size of DBMS, the characteristics of I/O access are changed by increasing the number of buffer cache misses. Therefore, changing the order of the tables to join so that the size of the outer table become larger than the buffer cache size, to investigate the tendency of

I/O accesses and create a model of I/O by linear regression. The I/O access tendency is shown in Figure 7.10.

The measured I/O response time (*io_response_time*) was 154 μs. The I/O cost of NLJ is as follows:

$$io\_cost = \left(A16 \times R_{I\_total(n)} + B16\right) \times io\_response\_time. \quad (7.25)$$

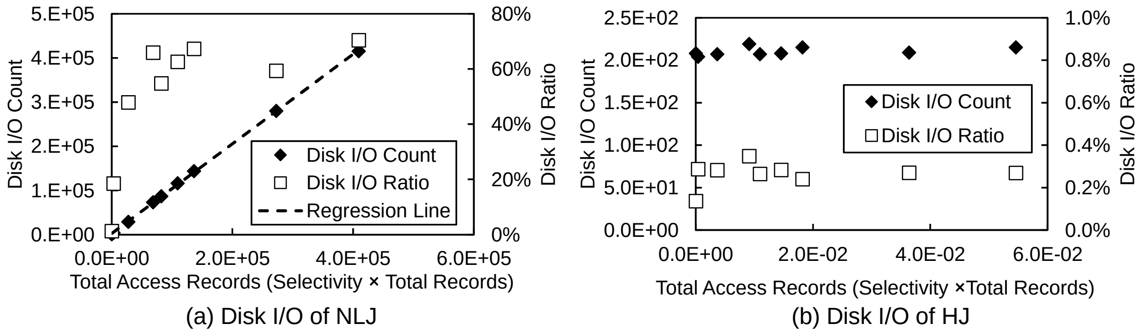


Figure 7.9.: Number of Disk I/Os and Disk I/O Processing Time Ratio while Joining Customer Table and Order Table by NLJ and HJ

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 7.9(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 1.1(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 7.11. The I/O cost of combination probe phase is as follows:

$$io\_cost = (A17 \times R_{I1} \times P_{I1} + A18 \times R_{I1} \times \sum_{i=2}^{n} rsk_i + B17) \times io\_response\_time \quad (7.26)$$

The regression models for the combination join are also expressed by the same equations, (7.7)–(7.21), as the two-table NLJ and HJ. Table 7.3 lists the definitions of the parameters.

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 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.1. The parameter setting of the cost calculation formulas was generated from the measurement values when joining Customer and Orders in Appendix A.2, whose size is SF5.

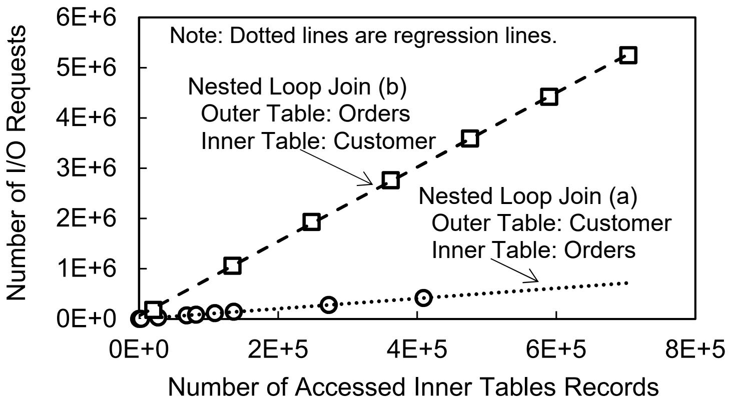
```
select count(*)
from
 customer
 STRAIGHT_JOIN orders
where
 c_mktsegment = 'MACHINERY'
 and c_acctbal > parameters
 and c_custkey = o_custkey
 and o_orderdate < date '1995-03-06';
```

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

parameter ∈ (9998, 9978, 9798, 9498, 9398, 9200, 9000, 8000, 7000)

(a) Nested loop Join Case 1

(b) Nested Loop Join Case 2



(c) Difference in Number of I/O Requests When Joining in the Different Order of Tables

Figure 7.10.: Difference of Number of Disk I/Os in Joining Table Size by NLJ

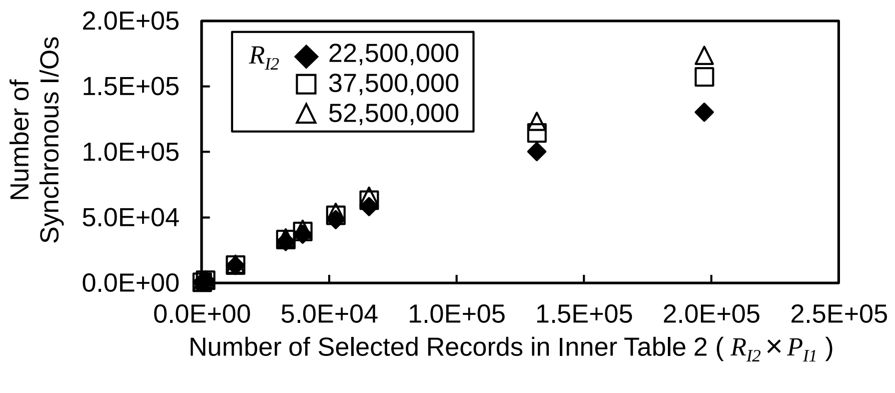


Figure 7.11.: Number of Synchronous Disk I/O Count during Combination Probe Phase

Table 7.2.: 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 7.3(1)
	A2	9.80×10^{-1}	B2	1.26×10^{7}	1.00	2.18×10^{-58}	Figure 7.3(2)
	A3	8.08×10^{-1}	B3	2.68×10^{6}	1.00	2.91×10^{-40}	Figure 7.3(3)
	A4	8.32×10^{-1}	B4	5.23×10^{4}	1.00	3.93×10^{-39}	Figure 7.3(4)
	A5	7.43×10^{-1}	B5	-1.18×10^{5}	1.00	3.77×10^{-34}	Figure 7.3(5)
	A6	2.58×10^{-1}	B6	1.18×10^{5}	9.98×10^{-1}	7.05×10^{-26}	Figure 7.3(6)
	A7	2.46×10^{4}	B7	4.63×10^{8}	9.99×10^{-1}	5.97×10^{-31}	Figure 7.3(7)
	A8	9.72×10^{-1}	B8	7.05×10^{6}	1.00	3.85×10^{-53}	Figure 7.3(8)
	A9	4.34×10^{-1}	B9	1.95×10^{6}	9.99×10^{-1}	5.17×10^{-28}	Figure 7.3(9)
	A10	9.44×10^{-1}	B10	-2.87×10^{4}	1.00	2.06×10^{-44}	Figure 7.310)
	A11	7.61×10^{-1}	B11	-4.84×10^{4}	1.00	2.80×10^{-35}	Figure 7.3(11)
	A12	2.39×10^{-1}	B12	4.84×10^{4}	9.98×10^{-1}	3.10×10^{-26}	Figure 7.3(12)
	A13	5.45×10^{-1}	B13	1.41×10^{8}	9.98×10^{-1}	1.21×10^{-25}	Figure 7.3(13)
	A14	8.59×10^{-1}	B14	-1.25×10^{9}	9.67×10^{-1}	9.00×10^{-15}	Figure 7.3(14)
	A15	2.90×10^{3}	B15	2.40×10^{7}	9.90×10^{-1}	1.35×10^{-19}	Figure 7.3(15)
HJ	A1	2.05×10^{3}	B1	1.58×10^{7}	1.00	4.21×10^{-40}	Figure 7.4(1)
Build	A2	9.88×10^{-1}	B2	2.53×10^{5}	1.00	2.19×10^{-61}	Figure 7.4(2)
	A3	9.71×10^{-1}	B3	-7.48×10^{4}	1.00	1.79×10^{-49}	Figure 7.4(3)
	A4	9.19×10^{-1}	B4	-6.57×10^{4}	9.99×10^{-1}	7.76×10^{-31}	Figure 7.4(4)
	A5	3.32×10^{-1}	B5	-1.64×10^{4}	9.29×10^{-1}	8.38×10^{-12}	Figure 7.4(5)
	A6	6.68×10^{-1}	B6	1.64×10^{4}	9.82×10^{-1}	4.49×10^{-17}	Figure 7.4(6)
	A7	6.10×10^{2}	B7	2.85×10^{5}	9.99×10^{-1}	4.46×10^{-30}	Figure 7.4(7)
	A8	9.90×10^{-1}	B8	-1.89×10^{3}	1.00	1.05×10^{-57}	Figure 7.4(8)
	A9	8.03×10^{-1}	B9	-2.39×10^{4}	1.00	6.00×10^{-33}	Figure 7.4(9)
	A10	9.04×10^{-1}	B10	1.58×10^{2}	1.00	8.94×10^{-46}	Figure 7.4(10)
	A11	2.13×10^{-1}	B11	-2.74×10^{3}	9.80×10^{-1}	1.13×10^{-16}	Figure 7.4(1)
	A12	7.87×10^{-1}	B12	2.74×10^{3}	9.98×10^{-1}	8.16×10^{-27}	Figure 7.4(12)
	A13	1.20	B13	-8.24×10^{6}	9.98×10^{-1}	2.14×10^{-25}	Figure 7.4(13)
	A14	3.69×10^{-1}	B14	2.02×10^{7}	1.00	6.83×10^{-32}	Figure 7.4(14)
	A15	2.94×10^{1}	B15	6.41×10^{5}	9.97×10^{-1}	2.86×10^{-24}	Figure 7.4(15)
HJ	A1	1.90×10^{3}	B1	2.33×10^{7}	1.00	3.46×10^{-46}	Figure 7.5(1)
Probe	A2	9.88×10^{-1}	B2	3.88×10^{5}	1.00	3.69×10^{-62}	Figure 7.5(2)
	A3	9.75×10^{-1}	B3	-1.76×10^{4}	1.00	6.81×10^{-52}	Figure 7.5(3)
	A4	8.13×10^{-1}	B4	-2.44×10^{4}	1.00	1.12×10^{-41}	Figure 7.5(4)
	A5	9.46×10^{-1}	B5	-2.44×10^{4}	1.00	8.30×10^{-36}	Figure 7.5(5)
	A6	5.45×10^{-2}	B6	2.44×10^{4}	9.56×10^{-1}	1.15×10^{-13}	Figure 7.5(6)
	A7	5.76×10^{2}	B7	-3.57×10^{7}	9.99×10^{-1}	6.14×10^{-27}	Figure 7.5(7)
	A8	9.89×10^{-1}	B8	-3.89×10^{5}	1.00	6.68×10^{-54}	Figure 7.5 (8)
	A9	7.29×10^{-1}	B9	1.58×10^{5}	9.88×10^{-1}	7.30×10^{-19}	Figure 7.5(9)
	A10	7.95×10^{-1}	B10	2.81×10^{4}	1.00	5.98×10^{-33}	Figure 7.5 (10)
	A11	9.34×10^{-1}	B11	-6.04×10^{4}	1.00	7.79×10^{-34}	Figure 7.5 (11)
	A12	6.60×10^{-2}	B12	6.04×10^{4}	9.52×10^{-1}	2.69×10^{-13}	Figure 7.5(12)
	A13	1.48	B13	2.87×10^{07}	9.87×10^{-1}	1.40×10^{-18}	Figure 7.5(13)
	A14	3.58×10^{-1}	B14	6.61×10^{7}	9.98×10^{-1}	1.75×10^{-25}	Figure 7.5(14)
	A15	3.45×10^{1}	B15	-1.39×10^{7}	9.35×10^{-1}	3.77×10^{-12}	Figure 7.5(15)
NLJ1	A16	1.02	B16	2.52×10^{3}	1.00	5.29×10^{-14}	Figure 7.10(c)-graph(a)
NLJ2	A16	7.40	B16	6.38×10^{4}	1.00	1.15×10^{-11}	Figure 7.10(c)-graph(b)
HJ	A16	0.000	B16	0.000	N/A	N/A	N/A

Note: NLJ1 is chosen in the case outer table size is smaller than the buffer cache size and NLJ2 is chosen in the other case.

Table 7.3.: 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 7.6(1)
	A2	9.86×10^{-1}	B16	1.79×10^{7}	1.00	1.11×10^{-44}	Figure 7.6(2)
	A3	8.53×10^{-1}	B3	6.58×10^{6}	9.94×10^{-1}	2.79×10^{-21}	Figure 7.6(3)
	A4	7.06×10^{-1}	B4	-6.80×10^{5}	9.99×10^{-1}	3.12×10^{-29}	Figure 7.6(4)
	A5	3.74×10^{-1}	B5	-1.58×10^{5}	9.99×10^{-1}	3.51×10^{-28}	Figure 7.6(5)
	A6	6.26×10^{-1}	B6	1.58×10^{5}	1.00	3.21×10^{-32}	Figure 7.6(6)
	A7	5.76×10^{2}	B7	2.31×10^{8}	9.75×10^{-1}	2.78×10^{-48}	Figure 7.6(7)
	A8	9.86×10^{-1}	B8	1.83×10^{6}	1.00	1.75×10^{-41}	Figure 7.6(8)
	A9	4.79×10^{-1}	B9	3.97×10^{6}	9.00×10^{-1}	2.03×10^{-10}	Figure 7.6(9)
	A10	9.36×10^{-1}	B10	-9.55×10^{5}	9.97×10^{-1}	7.85×10^{-25}	Figure 7.610)
	A11	3.70×10^{-1}	B11	-9.15×10^{4}	9.75×10^{-1}	6.97×10^{-16}	Figure 7.6(11)
	A12	6.30×10^{-1}	B12	9.15×10^{4}	9.91×10^{-1}	5.74×10^{-12}	Figure 7.6(12)
	A13	6.25×10^{-1}	B13	6.85×10^{8}	8.75×10^{-1}	1.51×10^{-9}	Figure 7.6(13)
	A14	4.49×10^{-1}	B14	-2.66×10^{8}	9.72×10^{-1}	2.16×10^{-15}	Figure 7.6(14)
	A15	4.93×10^{1}	B15	2.62×10^{7}	9.70×10^{-1}	1.03×10^{-45}	Figure 7.6(15)
Probe	A17	7.84×10^{-1}	B17	-6.59×10^{3}	9.79×10^{-1}	7.43×10^{-21}	N/A
(I/O)	A18	3.24×10^{-4}					N/A

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 (2.2)(2.3). The conventional method is expressed as the sum of the CPU cost and the I/O cost as mentioned in Section 2.2. 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 access path generated by MariaDB. In our measurement environment, the HJ access path for joining more than two tables is the combination case (Figure 4.2). 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 7.12 and Figure 7.13). The accuracy improvement ratios of the proposed method and conventional method are shown in Table 7.4. The second and third rows indicate the difference of selectivity between the intersection point of the measured result and that of the conventional method or proposed method. The accuracy improvement ratio is obtained by dividing the difference between the intersection selectivity of the conventional method and that of the proposed method by that of the conventional method in the equation 6.10. Table 7.4 shows that the proposed method improved the accuracy of selecting the proper join method by 90% or more.

As in chapter 6, we evaluated the updated conventional model case in which the unit

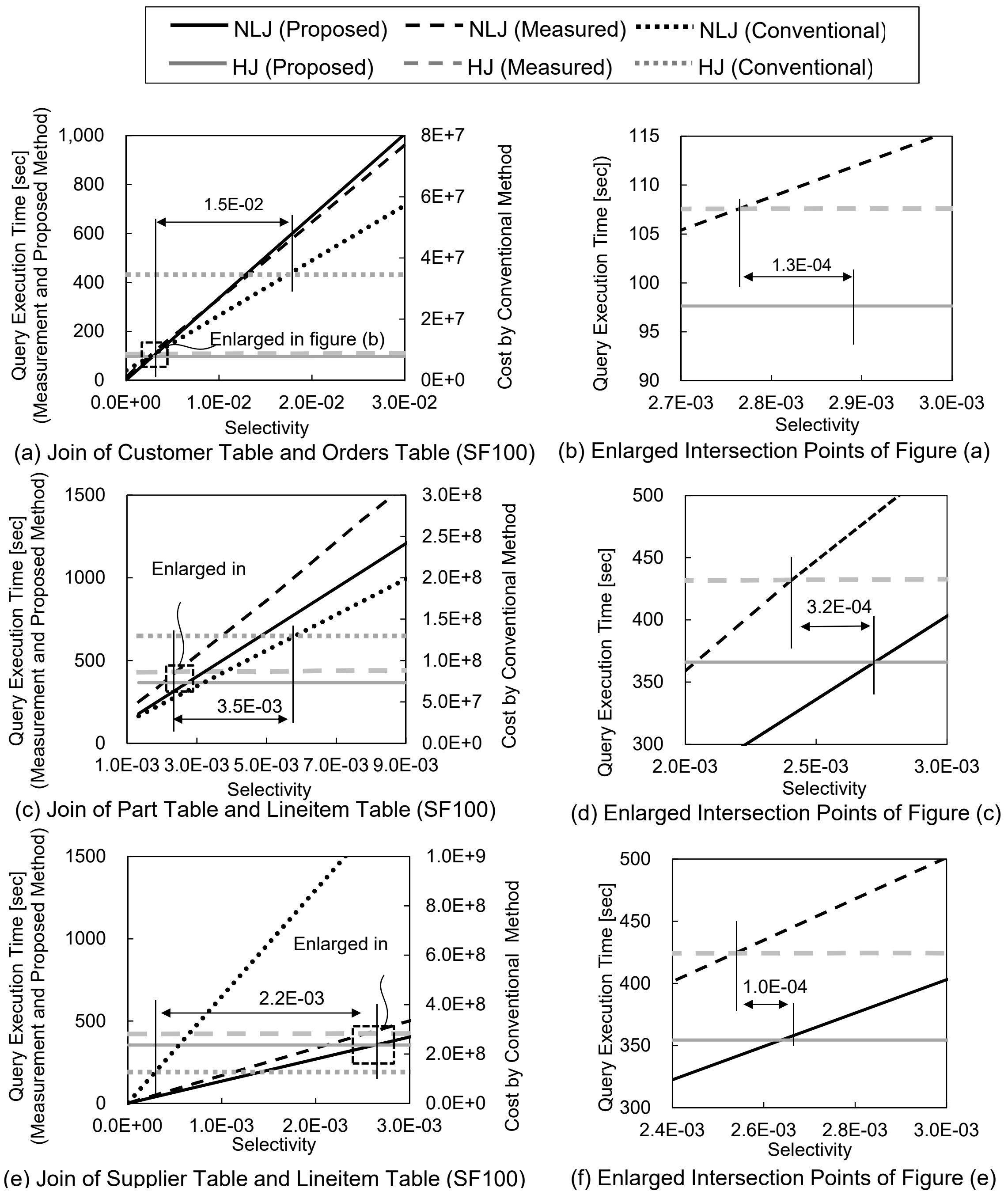


Figure 7.12.: Cost Comparison of Measured, Proposed Cost Model, and Conventional Cost Model Results for Joining Two Tables

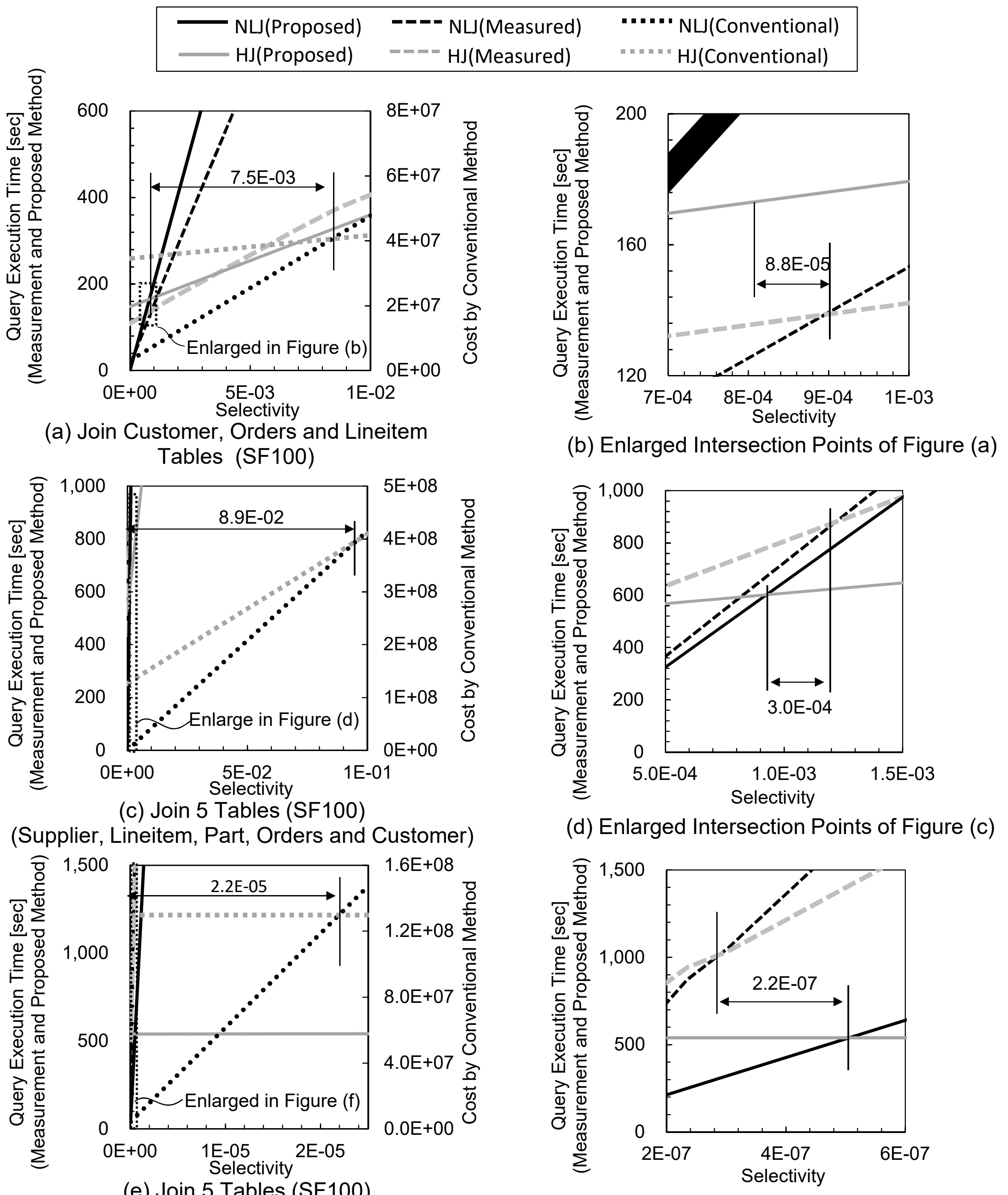


Figure 7.13.: Cost Comparison of Measured, Proposed Cost Model, and Conventional Cost Model Results for Joining Three or More Tables

Table 7.4.: Accuracy Improvement Ratio for Estimating Cross Point 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}	1.0×10^{-4}	8.8×10^{-5}	3.0×10^{-4}	2.2×10^{-7}
Accuracy Improvement ratio	99%	91%	95%	99%	97%	99%

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

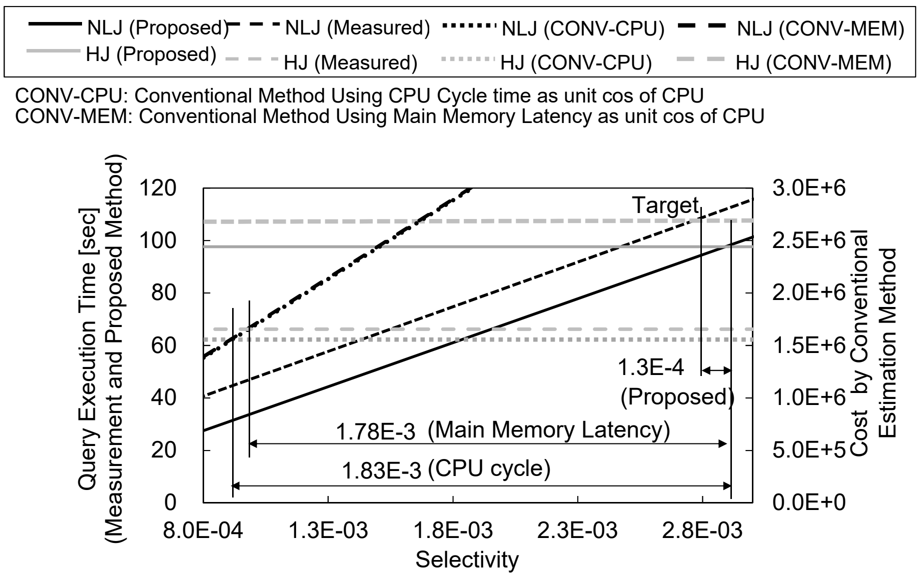
cost of CPU is the ratio of CPU cycle time and measured I/O latency ($CPR = 3 \times 10^{-6} = (1/2.13GHz)/154ms$) and the unit cost of CPU is the ratio of main memory latency[1] and measured I/O latency ($CPR = 6 \times 10^{-4} = 100ns/154ms$). The results of joining two tables (C-O, S-L and P-L) are shown in the Figure 7.14(a)(b)(c). The results of joining three or more tables (C-O-L, S-L-P-O-C and P-L-S-O-C) are shown in the Figure 7.15(a)(b)(c). The proposed method is estimated cross point more accurately than updated conventional method as shown in Table 7.5. The updated conventional models can calculate cost more accurately than the default cenventional method. However, the proposed model is more accurately than the updated conventional models in the evaluated join cases.

Table 7.5.: Accuracy Improvement Ratio for Estimating Cross Point of NLJ and HJ with Updated Conventional Method

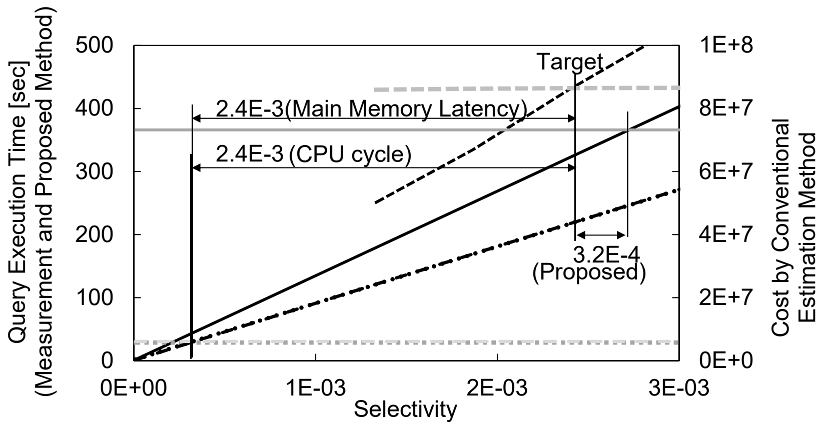
Join tables	C-O	P-L	S-L	C-O-L	S-L-P-O-C	P-L-S-O-C
CPU cost type	MEM	MEM	MEM	MEM	CPU	CPU
Conventional method	1.8×10^{-3}	2.4×10^{-3}	2.2×10^{-3}	4.4×10^{-4}	4.6×10^{-3}	8.6×10^{-7}
Proposed method	1.3×10^{-4}	3.2×10^{-4}	1.0×10^{-4}	8.8×10^{-5}	3.0×10^{-4}	2.2×10^{-7}
Accuracy Improvement ratio	93%	87%	95%	80%	94%	74%

Note1: C: Customer, O: Orders, L: Lineitem, P: Part, S: Supplier
Note2: CPU is *CPU cycle time*, and MEM is *main memory latency*.

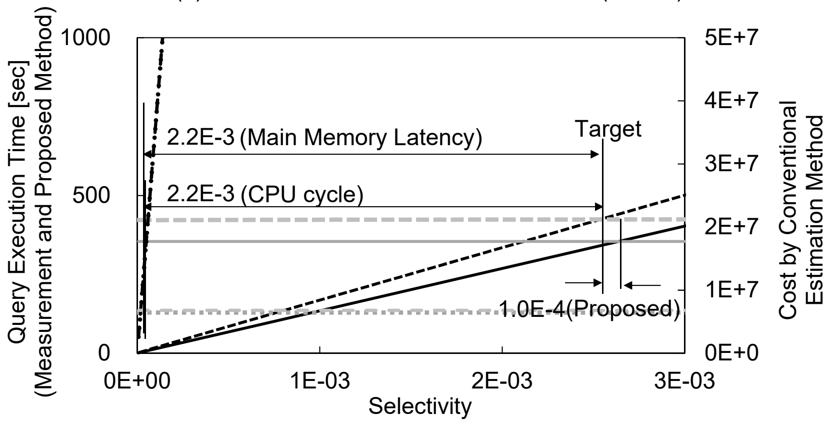
[1]It is not a measured value but an approximate value of main memory latency.



(a) Join of Customer Table and Orders Table (SF100)



(b) Join of Part Table and Lineitem Table (SF100)



(c) Join of Supplier Table and Lineitem Table (SF100)

Figure 7.14.: Cost Comparison of Measured, Proposed Cost Method and Updated Conventional Cost Method Results for Joining Two Tables

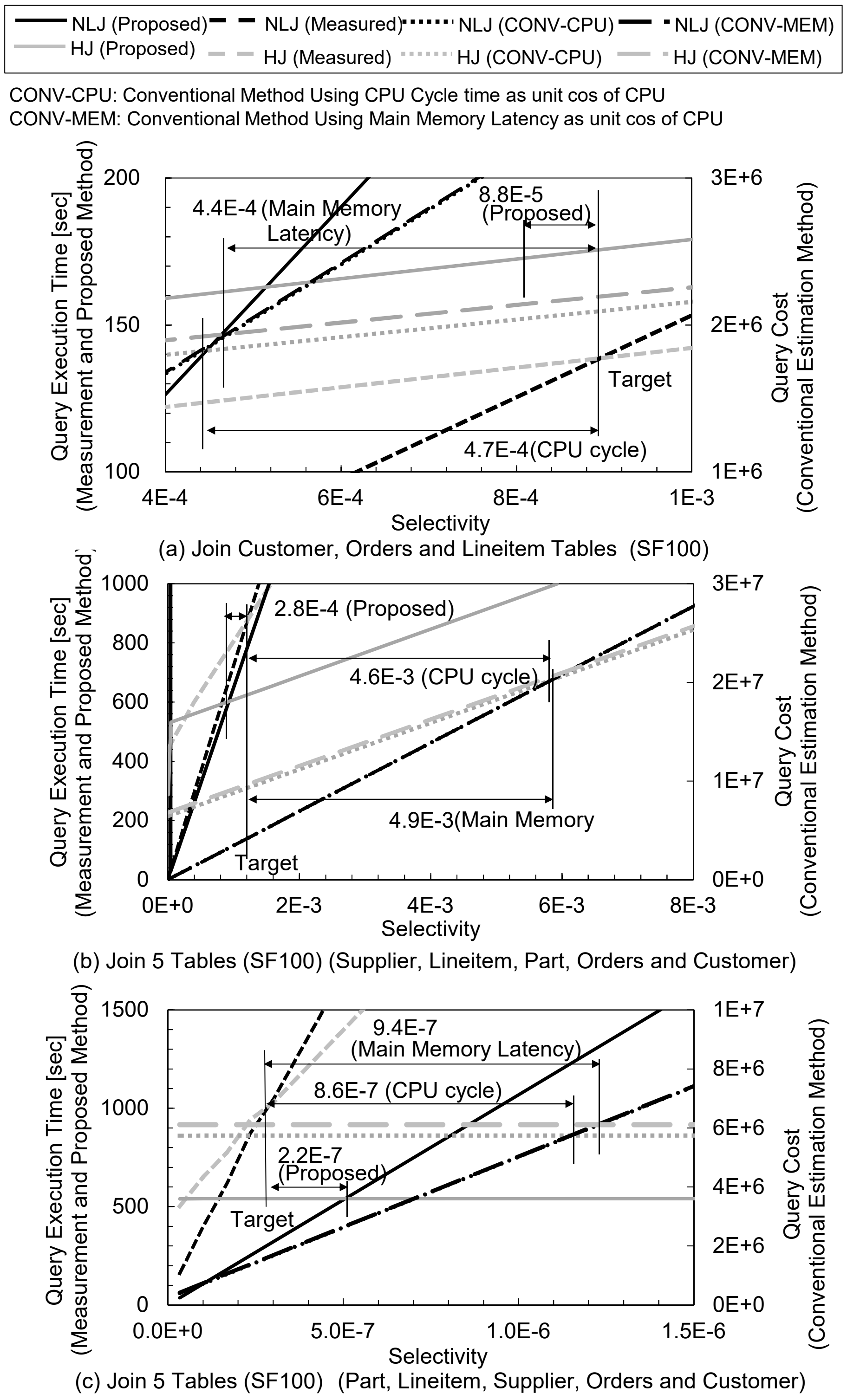
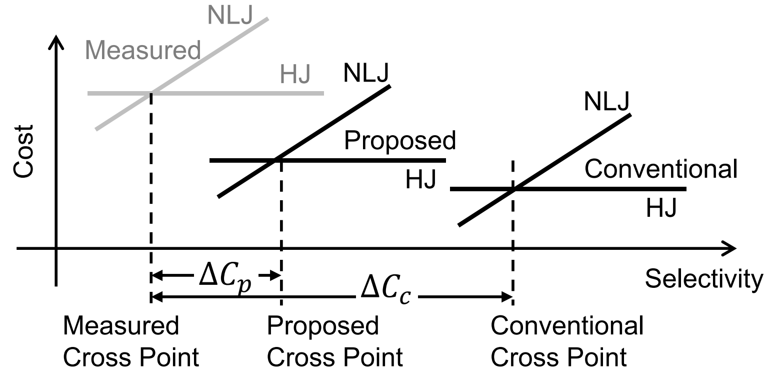


Figure 7.15.: Cost Comparison of Measured, Proposed Cost Method, and Updated Conventional Cost Method Results for Joining Three or More Tables

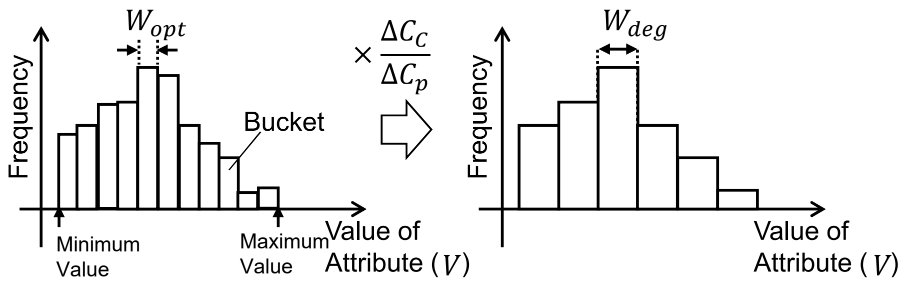
7.2. Degradation of Data Distribution Accuracy

When the prediction accuracy of cost is lowered, accuracy of selectivity can be lowered by using highly accurate cost calculation method. In other words, if the accuracy of cost can be degraded to the same degree as the accuracy of conventional cost calculation, the accuracy of the histogram representing the data distribution using the proposed cost calculation method can be degraded.

The cross point obtained by the proposed cost calculation method (hereinafter called "proposed cross point") is defined as C and the difference of measured cross point and proposed cross point is defined as ΔC_p in Figure 7.16(a). The difference of measured cross point and conventional cross point is defined as ΔC_c.



(a) Difference of Measured Cross Point and Proposed Cross point (ΔC)



(b) Equi-Width Histogram with Degraded Accuracy

Figure 7.16.: Histograms Using Difference of Measured Cross Point and Proposed Cross Point

We evaluated the accuracy of cost calculation methods by selectivity. The selectivity is related to the distribution of values of attributes. Therefore, we study the case that the sort parameter of histogram is value of an attribute. In the case of equi-width histogram and the attribute value V as a sorting parameter of histogram is a discrete value, the degraded bucket width W_{deg} is given by

$$W_{deg} = \frac{\Delta C_c}{\Delta C_{opt}} \times W_{opt}, \tag{7.27}$$

where the minimum bucket width is W_{opt} in Figure 7.16(b).

The equation 7.27 means that the accuracy of histogram can be degraded by the ratio of $\Delta C_{opt}/\Delta C_c$. When the bucket width is W ($W < W_{deg}$), it is possible to widen the bucket width to $W_{updated}$ in equation 7.28 using the proposed cost calculation method. The bucket expansion rate α is given by

$$\alpha = \frac{W_{updated}}{W} = \frac{\Delta C_c}{\Delta C_{opt}} \times \frac{W_{opt}}{W}. \tag{7.28}$$

If we allow the same accuracy of cost using the proposed cost calculation method as that of the conventional method, we can use the histogram with the bucket width $\alpha \times W$.

If the cross point in advance can be identified, the above idea can also be applied to the equi-depth histogram.

As a result of evaluating the ratio of increasing the width of the bucket size in the example used in this study (Table 7.6), it is found that the bucket width can be expanded up to four times of W_{opt}.

Table 7.6.: Increasing Ratio of Equi-Width Histogram buckets Width Using Updated Conventional Method and Proposed Method

Join tables	C-O	P-L	S-L	C-O-L	S-L-P-O-C	P-L-S-O-C
$\Delta C_c/\Delta C_{opt}$	14	8	22	5	16	4

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

7.3. Verification of Determining Join Method and Ordering Joining Tables

7.3.1. Verification Method

In this section, using the proposed method, we verify whether the join method and the order of joining tables that minimizes execution time are found. First, we collect statistical information of CPU and create cost formula in Section 7.1. Then, in two or more joins in the same database environment, join the tables in all executable order, and compare the execution time with the cost calculation result. Similarly to the above experiments, we used TPC-H database. The combinations of TPC-H tables to join are as follows: Customer and Orders, Part and Lineitem, Supplier and Lineitem, Customer, Orders and Lineitem, and Supplier, Lineitem, Orders, Customer and Part. The queries for join operation is shown in Figure 7.17. When measuring query execution time, the order of tables to join is fixed by `STRAIGHT_JOIN` hint.

The CPU statistical information for formulating the cost calculation is measured through the reference CPU, the details of which are provided in Table 7.7. The prediction of the cross point of two join methods by using the cost calculation formula is aimed at determining case SF100 on the Skylake-based processor as shown in Table 7.7. The cost calculation formula

```
select count(*)
from  lineitem, part
where
  (p_type='STANDARD ANODIZED TIN’
  or p_type='STANDARD ANODIZED STEEL’)
  and p_size < 2
  and p_partkey = l_partkey
  and l_shipdate < date '1995-03-06';
```

(a) Join of Lineitem and Part

```
select count(*)
from  lineitem, supplier
where
  s_acctbal > 9000
  and s_nationkey = 0
  and s_suppkey = l_suppkey
  and l_shipdate > date '1995-03-06';
```

(b) Join of Lineitem and Supplier

```
select count(*)
from  orders, customer
where
  c_mktsegment = 'MACHINERY’
  and c_acctbal > 9000
  and c_custkey = o_custkey
  and o_orderdate < date '1995-03-06';
```

(c) Join of Orders and Customer

```
select count(*)
from  customer, orders, lineitem
where
  c_mktsegment = 'MACHINERY’
  and c_acctbal > 9000
  and c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate < date '1995-03-06’
  and l_shipdate > date '1995-03-06’;
```

(d) Join of Customer, Orders, and Lineitem

```
select count(*)
From customer, orders, lineitem, part, supplier
where
  l_shipdate > date '1995-03-06' and l_quantity = 10 and  l_discount=0.08
  and p_size < 2
  and o_orderdate < date '1995-03-06‘
  and c_acctbal > 9000
  and s_acctbal > 9000
  and s_suppkey = l_suppkey
  and l_partkey = p_partkey
  and l_orderkey = o_orderkey
  and o_custkey = c_custkey;
```

(e) Join of Part, Supplier, Customer, Orders, and Lineitem

Figure 7.17.: Queries for Evaluating Join Method and the Order of Joining Tables

of Skylake was generated in Section 7.1. As we used a disk-based DBMS, the join cost is the sum of the CPU and I/O costs. The formula to determine the I/O cost was estimated using the measured value.

Table 7.7.: Evaluation Environment

CPU	Xeon E3-1230 v5 3.4 GHz 4-core, LLC 8 MB [Skylake] ×1
Memory	DDR4 32 GB
Disk for DB	PCIe NVMe Flash SSD 800 GB
OS/DBMS	CentOS 6.6 (×64)/MariaDB 10.1.8 with InnoDB
DB size	TPC-H SF5 (5GB) for generation of cost calc. formula and SF100 (100 GB) for validation
Measuring tool	[CPU events] VTune Amplifier, [Query, I/O] System Tap

7.3.2. Comparison of Measured Query Execution Time and Estimated Query Cost

Table 7.8 shows measured time and estimated cost using our proposed cost calculation method excluding the cases where the join key does not exist between tables. In the case of joining two or three tables, it was confirmed that the join order of tables and the join method whose query processing time was minimum execution time in the measurement are the same as those of the smallest cost obtained by our proposed cost calculation method.

However, in the case of joining five tables, the predicted minimum join cost cases were L→P→S→O→C, L→P→O→S→C and L→P→O→C→S. The smallest measurement result case was S→L →O→P →C. The difference between the minimum execution time and the execution time in the predicted minimum join order is seven seconds, that is 2.9% of minimum execution time.

In the case of joining three tables, the case of joining in order of Orders and Lineitem and the case of joining in order of Lineitem and Orders are not consistent with the measured. In the case of joining five tables, except for the cases starting with the join of Orders and Lineitem (O-L Join), the magnitude relation between the cost of NLJ and the cost of HJ by the proposed cost calculation method is consistent with the measured value. The reason why the predicted cost of O-L Join by NLJ is far from the measured value is that the operation of the I/O of the DBMS used in this study is different from the other cases. Figure 7.18 shows the number of issued I/O requests while joining by NLJ and Figure 7.19 shows the distribution of I/O request size. From these figures, in the case starting with the join of Lineitem and Orders (L-O Join) and the case of O-L Join, issued I/O request size is larger than that of the other cases. However, the number of I/ O requests of L-O join and O-L Join case is smaller than the other case. This suggests that the cost models of the CPU and I/O have to be changed because the behavior of the DBMS is different from the queries that is used for creating the proposed cost calculation model.

In addition, we examined the relationship between the number of I/O requests and selectivity of joining Orders and Lineitem to investigate whether behavior of the DBMS changes.

Table 7.8.: Measured Time and Estimated Cost to Decide a Join Method and Order of Accessing Tables

Join Order	Query execution time and cost (second)			
	Measured time		Estimated cost	
	NLJ	HJ	NLJ	HJ
C → O	489	$\boxed{61}$	488	$\boxed{59}$
O → C	11163	179	84001	61
S → L	483	$\boxed{241}$	391	$\boxed{213}$
L → S	1027	329	318302	223
P → L	$\boxed{54}$	244	$\boxed{29}$	220
L → P	42626	832	318302	230
C → O → L	910	$\boxed{264}$	1672	$\boxed{267}$
L → O → C	1217	4506	101667	5133
O → C → L	11349	426	14900	457
O → L → C	1222	6774	61357	19463
S → L → O → P → C	11864	$\boxed{245}$	11432	416
S → L → O → C → P	11878	248	11432	416
P → L → O → S → C	2693	249	2516	419
P → L → S → O → C	2693	250	2515	419
P → L → O → C → S	2674	250	2516	419
S → L → P → O → C	11908	251	11427	416
L → P → S → O → C	361	252	731	$\boxed{261}$
L → P → O → S → C	367	252	737	$\boxed{261}$
L → P → O → C → S	363	253	737	$\boxed{261}$
L → O → S → P → C	345	305	1087	359
L → O → S → C → P	343	305	1090	361
L → O → P → C → S	346	305	1063	359
L → O → C → P → S	346	305	1064	361
L → O → P → S → C	343	305	1063	359
L → O → C → S → P	340	305	1089	359
O → L → S → C → P	366	5869	60602	5857
O → L → P → S → C	369	5865	60598	5850
O → L → S → P → C	372	5879	60602	5857
O → L → P → C → S	372	5854	60598	5850
O → L → C → P → S	372	5852	60553	5850
O → L → C → S → P	373	5867	60597	5850
C → O → L → P → S	4331	488	1673	1098
C → O → L → S → P	4343	488	1673	1102
O → C → L → S → P	11615	1322	6057	718

Note1: The minimum query execution time and minimum query cost are surrounded by frames.
Note2: C: Customer, O: Orders, L: Lineitem, P: Part, S: Supplier

Figure 7.20 shows that the behavior of I/O is changed by selectivity. Therefore, it is possible to further improve the accuracy of the cost calculation model by creating a cost model consistent with the internal operation policy of the DBMS.

The difference between the actual and the predictions in the multi-table join was found to be that the DBMS's I/O is one of the reasons for the actual difference between the model. The DBMS used in this experiment generated the same access path regardless of selectivity. Therefore, the operation of the I/O engine changed with selectivity. It is necessary to consider the model of the I/O engine as a future work.

On the other hand, when the first table to join was Orders table or Lineitem table, the join method in which the actual execution time was minimum was different from the join method in which the estimated cost was minimum. Looking into the details, it was found that there is a problem in the accuracy of the estimated I/O processing cost. In this study, we introduced a behavior model of CPU pipeline to improve the accuracy of CPU cost. To improve the accuracy of I/O, a precise I/O behavior model like CPU will be introduced as future works.

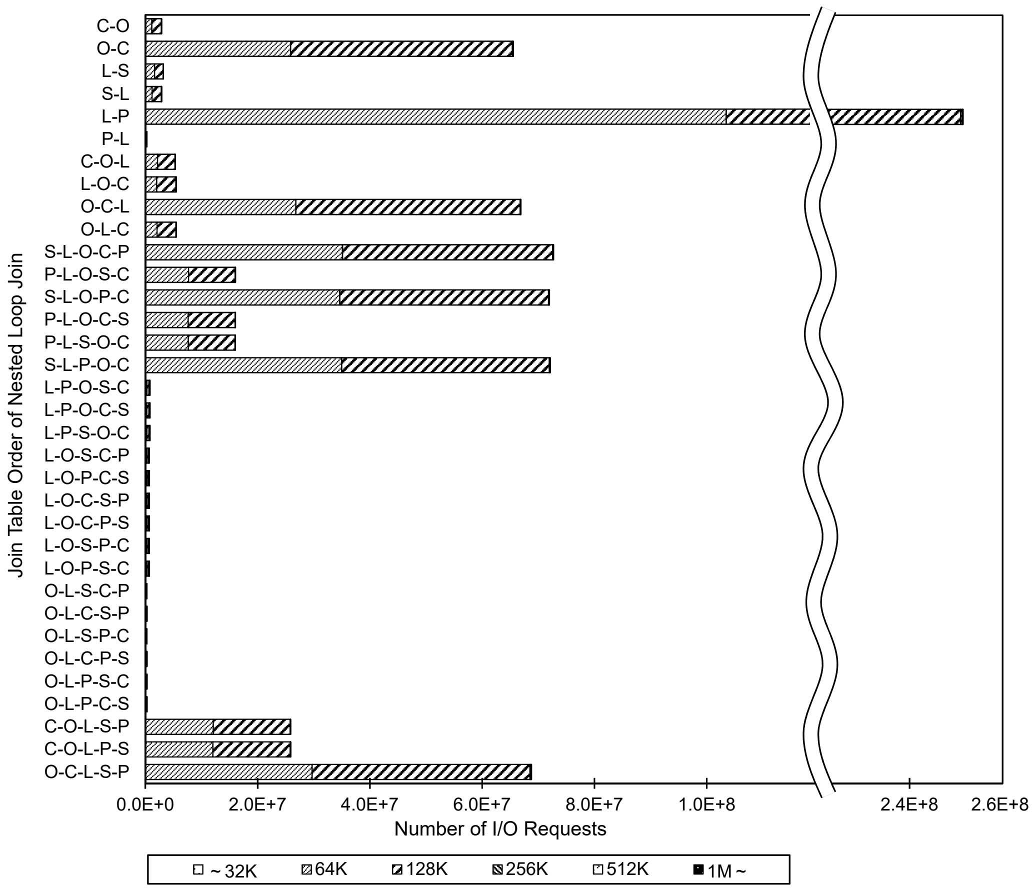


Figure 7.18.: Distribution of Number of I/O Requests in Nested Loop Join

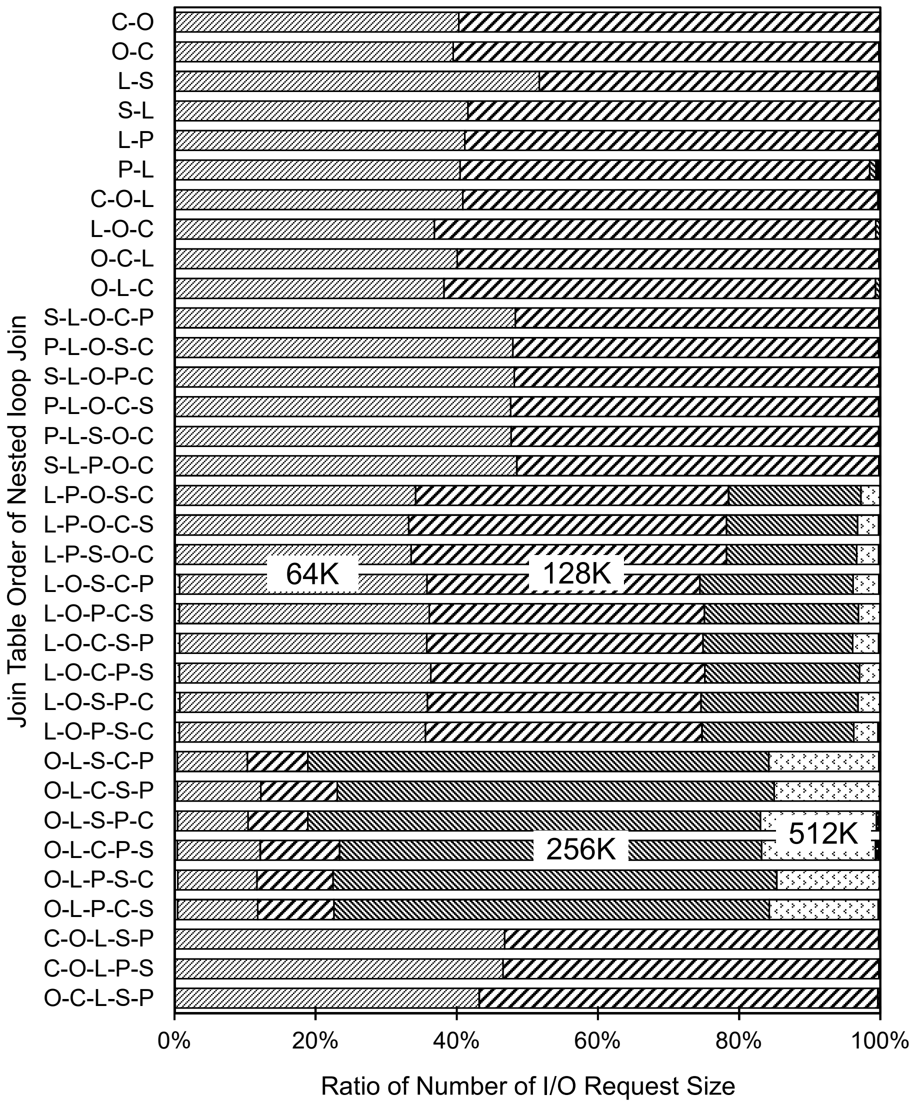
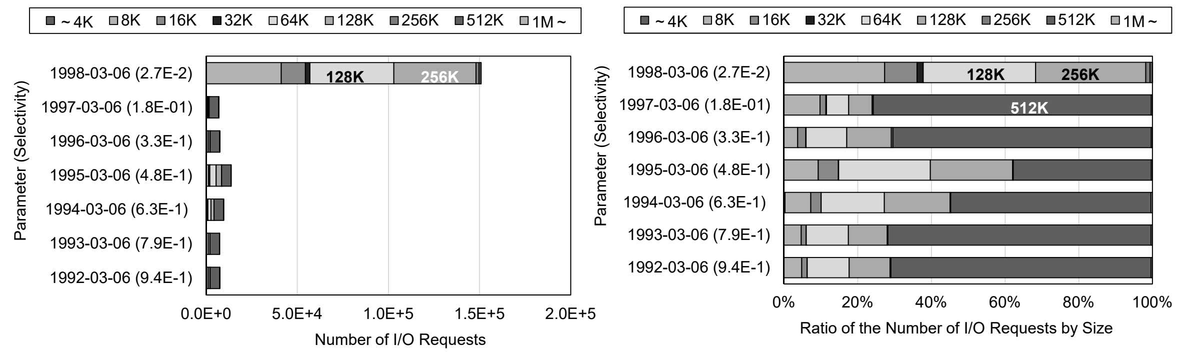


Figure 7.19.: Ratio of Size of Issued I/O requests in Nested Loop Join

```
select count(*)
from
  orders
  STRAIGHT_JOIN lineitem
where
  l_orderkey = o_orderkey
  and o_orderdate < date parameter
  and l_shipdate > date '1995-03-06';
```

parameter ∈ ('1992-03-06', '1993-03-06', '1994-03-06', '1995-03-06' , '1996-03-06', '1997-03-06', '1998-03-06')

(a) Query of Joining Two Large Tables, Orders and Lineitem



(b) Difference in I/O Request Count and Size when Joining Two Large Tables

Figure 7.20.: I/O Request Size in Joining Orders and Lineitem by Nested Loop Join

7.4. 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 [56, 57], 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 4.2. Theoretically, there is a case in which HJ is assigned after NLJ. In the case where HJ was executed after NLJ (Figure 7.21), 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 4.1, 4.2, and 7.21, 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.

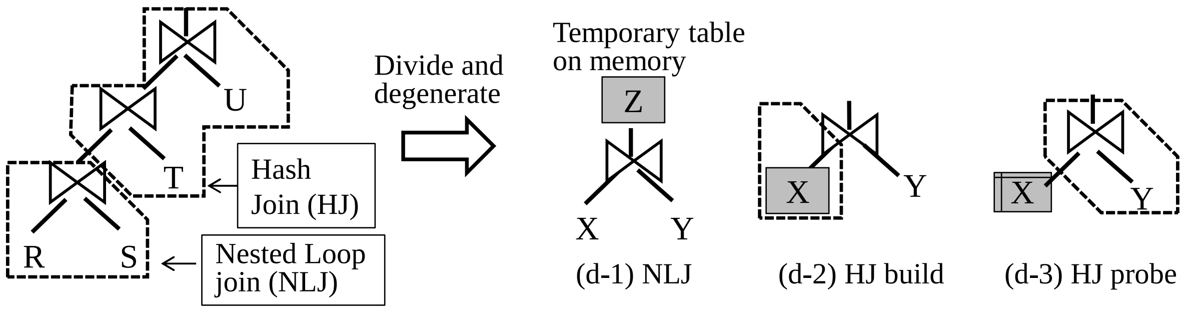


Figure 7.21.: NLJ after HJ Case

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-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 (7.1).

7.5. Conclusions

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 reduced by 71% to 94% of 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.

8. Updating Cost Calculation Method Applying to Different Generation CPUs

In developing our proposed cost calculation formula, reference queries are executed on a small reference database (Figure 8.1). On executing the queries, the performance monitor installed in the CPU measures CPU events such as the number of executed instructions and the number of cache hits. The cost calculation formulas are created using those measurement values. Analytic queries are executed according to the query access path with the smallest cost, calculated using the cost calculation formulas. The measurement time of the CPU events can be shortened by using the small-scale data because the execution time is proportional to the size of the database. This makes it possible for the database administrator to quickly provide the most suitable database environment to the user.

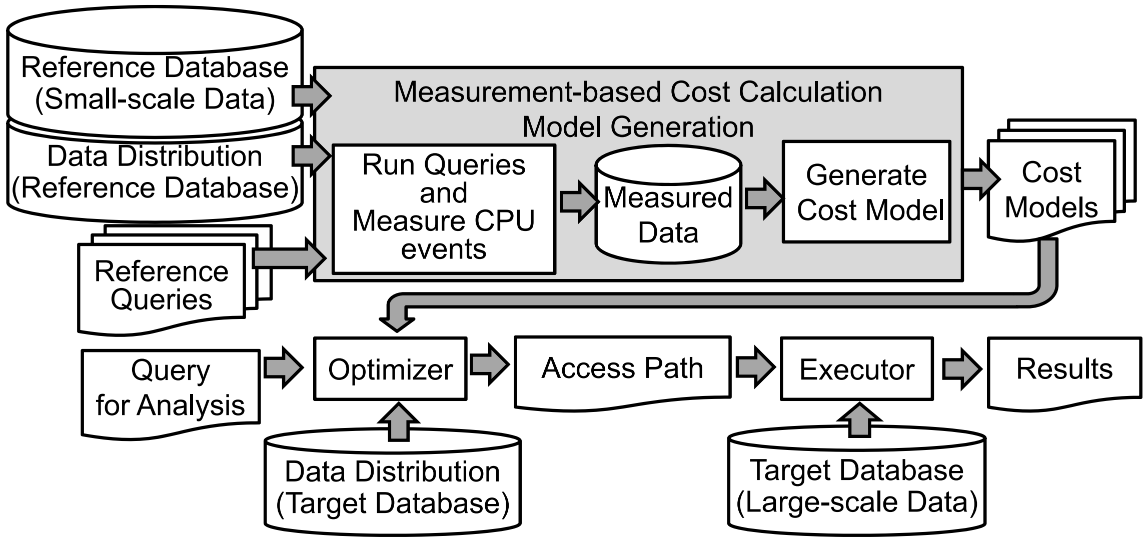

Reference Database (Small-scale Data)
Data Distribution (Reference Database)
Reference Queries
Measurement-based Cost Calculation Model Generation
Run Queries and Measure CPU events
Measured Data
Generate Cost Model
Cost Models
Query for Analysis
Optimizer
Access Path
Executor
Results
Data Distribution (Target Database)
Target Database (Large-scale Data)

Figure 8.1.: Overview of Query Execution Flow Using Measurement-based Cost Calculation

However, the regression approaches we took in Chapter 6 and 7 need to re-measure the statistic information and recreate the regression model whenever the CPU is upgraded. In other words, the operational cost of the database should be increased. In a cloud environment especially, it is unrealistic to recreate the cost model every time the VM with the database is transferred to another server using CPUs with a different architecture. This study aims to verify whether CPUs with different architectures can be employed with no changes or minor changes to the measurement-based cost calculation method. The change in performance as a result of architectural changes, such as the memory latency and cache size, among different generations of CPUs is reflected in the measurement-based cost calculation formula for join processing. We used this updated cost calculation formula to verify whether it is possible to select the joining methods, that is, nested loop join (NLJ) and hash join (HJ), accurately for different generations of CPUs. We determined that the updated cost calculation formulas are able to estimate the cross point accurately. In conclusion, our proposed updating method for the measurement-based join cost calculation allows portability of the join cost calculation across different generations of CPUs and can contribute to reducing the cost of cloud service platforms.

8.1. Cost Calculation Method for Join Operation

When executing a query, the DBMS creates multiple access paths for query processing and estimates the processing cost for each plan. The minimum cost access path is selected from a plurality of candidates. For example, when the DBMS joins two tables, such as the R and S tables shown in Figure 1.2(a), it generates the access path illustrated in Figure 1.2(b) and estimates the cost of each join method by using statistical information such as the distribution of the join key value and it selects the method with the lowest cost. In general, the cost calculation is formulated using the sum of CPU and I/O costs, as follows:

$$cost = CPU\_cost + IO\_cost. \tag{8.1}$$

Moreover, the CPU cost (CPU_cost) is obtained with respect to the number of CPU cycles (C_{CPU}) and CPU frequency ($CPU_{frequency}$) in Equation (8.2). The I/O cost (IO_cost) is obtained through the product of the number of I/Os (M_{IO}) and the I/O response time ($IO\_response\_time$) in Equation (8.3).

$$CPU\_cost = \frac{C_{CPU}}{CPU_{frequency}} \tag{8.2}$$

$$IO\_cost = M_{IO} \times IO\_response\_time \tag{8.3}$$

In general, the cost of a join operation is obtained as a function of the ratio of the number of extracted records to the total records. This ratio is known as the *selectivity*. In Figure 1.2, the selectivity is determined according to condition x for column R.C in Figure 1.2(c). Two cost functions intersect at X_{cross}. Join Method 2 must be selected from the left side of X_{cross}, and join Method 1 should be selected from the right side of X_{cross}. If the DBMS cannot estimate the selectivity X_{cross} accurately, it will likely select the wrong join method.

In Chapter 7, we proposed a measurement-based join CPU cost calculation method to obtain the selectivity of cross point X_{cross} accurately. We focused on the CPU pipeline operation to construct an accurate model for calculating the cost of the join operation. The total number of execution cycles during the join operation is composed of the cache miss penalty on the instruction access cycles $C_{ICacheMiss}$, branch misprediction recovery cycles C_{MP}, and data access cycles $C_{DCacheAcc}$. The join cost C_{Join} is the sum of these cycles:

$$C_{Join} = C_{DCacheAcc} + C_{ICacheMiss} + C_{MP}, \tag{8.4}$$

where $C_{ICacheMiss}$, $C_{DCacheAcc}$, and C_{MP} are given by the linear regression equation representing the number of cache hits, main memory accesses, and branch mispredictions measured through the performance monitor embedded in the CPU. $C_{ICacheMiss}$ is expressed by the linear regression equation of the sum of products of the number of cache hits M_{LiI_hit} $(i = 2, \cdots, N)$ or main memory accesses M_{MMI_acc} in the instruction accesses and the latency of the cache L_{LiI}, $(i = 2, \cdots, N)$ or of the main memory L_{MMI} in Equation (8.5). Here, w_I and b_I are the slope and intercept, respectively, of the linear regression of the measured value of $C_{DCacheAcc}$ versus the sum of the products of the number of cache or main memory reads and the latency. Similarly, $C_{DCacheAcc}$ is a linear regression equation that expresses the sum of the products of the number of cache hits M_{LiD_hit} $(i = 1, \cdots, N)$ in terms of data or main-memory accesses M_{MMD_acc}, considering the number of data accesses and latency of the cache L_{LiD} $(i = 1, \cdots, N)$ or main memory L_{MMD} in Equation (8.6). Here, w_I and b_I are defined similarly as in Equation (8.5). In this case, w_D and b_D are the slope and intercept, respectively, of the linear regression of the measured $C_{DCacheAcc}$ versus the sum of products of the number of cache or main memory reads and latency. C_{MP} is given by the measured M_{MP}, number of accessed records R, w_{MP}, and b_{MP} in Equation (8.7). Here, w_{MP} and b_{MP} are the slope and intercept, respectively, of the linear regression line of the measured C_{MP} versus the number of branch mispredictions M_{MP}.

$$C_{ICacheMiss} = w_I \times (\sum_{Li=L2}^{LN} (M_{LiI\_hit} \times L_{LiI}) + (M_{MMI\_acc} \times L_{MMI})) + b_I \tag{8.5}$$

$$C_{DCacheAcc} = w_D \times (\sum_{Li=L1}^{LN} (M_{LiD\_hit} \times L_{LiD}) + (M_{MMD\_acc} \times L_{MMD})) + b_D \tag{8.6}$$

$$C_{MP} = w_{MP} \times R + b_{MP} \tag{8.7}$$

The numbers of cache hits and main memory accesses are given by the linear regression equation for the numbers of cache misses and executed instructions. Moreover, the number of executed instructions is given by a linear regression equation for the number of records accessed during the join operation. The numbers of cache memory hits and main memory accesses are given by

$$M_{L1dt\_hit} = w_{L1dt\_hit} \times I_{tp} + b_{L1dt} \tag{8.8}$$

$$M_{Lidt\_hit} = w_{Lidt\_hit} \times (I_{tp} - \sum_{j=1}^{i-1} M_{Ljdt\_hit}) + b_{Lidt} \tag{8.9}$$

$$M_{MMdt\_acc} = I_{tp} - \sum_{j=1}^{N} M_{LNdt\_hit} \tag{8.10}$$

where dt is the data type accessed by the CPU, namely an instruction (I) or data (D), tp is the instruction type, namely all types of instructions *all* or data load instructions *load*, w_{Lidt_hit} and w_{MMdt_hit} are the slopes of the linear regressions, and b_{Lidt} and b_{MMdt} are respectively the slope and intercept of the linear regression. The coefficients of the cost calculation formulas are created from the measurement results of the CPU.

8.2. Extension of Cost Calculation Method

To apply the above-mentioned cost calculation to the cloud environment in Figure 1.3, we must solve two problems. The first problem is to determine how to apply the cost calculation formulas to different generations of CPUs for a single VM running on a physical server. The second problem is to determine how to apply the formulas to multiple VMs running on a physical server concurrently.

We attempted to solve the first problem by categorizing the architectural changes when the CPU generation changed, and reflecting those changes in the coefficients of the cost calculation formulas. Table 8.1 lists the effect of the classification of the architectural change on the CPU cost of Equation (8.2) and the three elements ($C_{ICacheMiss}$, $C_{DCacheAcc}$, and C_{MP}) of the cost calculation formula of Equation (8.4). Changing the number of CPU cores affects the parallel processing of the table scan of the HJ operation. As most of the DBMS products, such as Oracle, DB2, and SQL Server, can scan tables in parallel, the CPU cost of HJ is reduced by 1/(the number of CPU cores). However, as many current open source DBMS cannot scan in parallel, the cost calculation formula is not affected.

Table 8.1.: Architectural Difference between CPU Generations

Component	Difference	Impact on cost calculation
CPU core	Frequency	Total CPU cost
	Number of cores	Total CPU cost
Cache	Size	Number of cache hits
	Latency	Instruction or data access latency
	Associativity	Number of cache hits
Main memory	Latency	Instruction or data access latency
Branch Predictor	Enhancement of accuracy	Branch misprediction penalty

The architectural changes related to the measurement-based join cost calculation formulas are size, the associativity and latency of the cache memory, latency of the main memory, and branch prediction penalty [58, 59]. Changes in the cache size and associativity affect the number of cache hits. Further, changes in the latency of the cache or main memory are already parameterized in the join cost calculation formulas. We introduce *the ratio of level-i cache miss ratio changes* MC_{Li_dtype} to *the ratio of latency change* LC_{mem_dtype} to apply the join cost calculation by using the values measured on a *reference CPU*, which is the CPU used for measuring the parameters of the cost calculation formula. The CPU used to estimate the join cost is referred to as the *target CPU*.

The numbers of cache memory hits and main memory accesses of the target CPU are given by

$$M_{L1dt\_hit} \text{ of target CPU} = MC_{L1\_dtype} \times (w_{L1dt\_hit} \times I_{tp} + b_{L1dt}) + (1 - MC_{L1\_dtype}) \times I_{tp}, \tag{8.11}$$

$$\begin{aligned} M_{Lidt\_hit} &\text{ of target CPU} \\ &= MC_{Li\_dtype} \times (w_{Lidt\_hit} \times (I_{tp} - \sum_{j=1}^{i-1} M_{Ljdt\_hit} \text{ of target CPU}) + b_{Lidt}) \\ &+ (1 - MC_{Li\_dtype}) \times (I_{tp} - \sum_{j=1}^{i-1} M_{Ljdt\_hit}), \end{aligned} \tag{8.12}$$

$$M_{MMdt\_acc} \text{ of target CPU} = I_{tp} - \sum_{j=1}^{N} M_{LNdt\_hit} \text{ of target CPU} \tag{8.13}$$

LC_{mem_dt} is defined as

$$LC_{mem\_dt} = \frac{L_{mem\_dt} \text{ of target CPU}}{L_{mem\_dt} \text{ of reference CPU,}} \tag{8.14}$$

where *mem* is level i $(1 \leq i \leq N)$ of the cache memory (Li) or main memory (MM), and dt is the data type accessed by the CPU, namely instruction (I) or data (D). mem_dt is either LiI, MMI, LiD, or MMD.

The memory latency of the target and reference CPUs can be obtained using a pointer chasing-type micro-benchmark [48, 55]. To obtain CRH_{mem_dt}, the cache hit/miss ratio for various cache sizes and associativity on the same CPU is required. The relationship between the cache hit/miss ratio and size and associativity can be obtained through a simulation [60,61]. Alternatively, the cache miss rate of the target CPU can be obtained by applying the rule stating that the cache miss rate decreases as a power law of the cache size [62].

BM is defined as the ratio of the number of branch misprediction penalties of the target CPU to those of the reference CPU in Equation (8.15). This led us to propose the use of a CPU simulator [63] and the approximation of the number of pipeline stages measured using the micro-benchmark [55] to obtain the branch misprediction penalty for the target and reference CPUs.

$$BM = \frac{\text{Branch misprediction penalty of target CPU}}{\text{Branch misprediction penalty of reference CPU}} \tag{8.15}$$

When splitting the table scan of the hash join into multiple CPU cores, the number of CPU cores is included in the cost calculation formula. For example, in a hash join for a commercial

DBMS, the table is divided into N_{core} CPU cores defined by the database administrator and a table scan is performed. In this case the CPU cost ($cost_{Nparallel}$) is given by

$$cost_{Nparallel} = (CPU\_cost + IO\_cost)/N_{core}. \tag{8.16}$$

When N_{core} changes for different generation CPUs, not only must Equation (8.16) be applied, but also the number of cache hits and the memory latency change must be considered. In the case where CPU cores belonging to the same CPU chip, or when simultaneous multithreading (SMT), a plurality of threads executing simultaneously on a single CPU core is used. This results in the decrease of the effective capacity of the LLC allocated to each CPU core. Therefore, the number of cache misses increases.

Many of Intel's processors use the inclusive cache architecture [52], so as instructions and data are removed from the shared L3 cache when accessing large-scale data increases, the same instructions and data existing in the L1 cache and L2 cache are removed also. As a result, the number of cache misses increases. Furthermore, owing to the increasing number of cache misses and memory access requests, access concentration in the main memory occurs, and latency may increase. To deal with the case of parallel processing in the measurement-based cost calculation method, it is necessary to measure the CPU events under parallel processing when N_{core} is changed.

In addition, when multiple DBMS instances run on a single physical server and queries are executed on each DBMS instance concurrently, or when multiple VMs are running on a single physical server and queries are executed concurrently on these VMs, the number of cache misses increases and memory latency increases.

In this study, as the first step in studying the cost calculation method for cloud database services, we focused on a single DBMS instance running on a single VM on a physical server with different-generation CPUs. However, as a single server running multiple VMs is a typical system configuration, we considered the case of multiple VMs as follows.

When many VMs are running concurrently, two problems occur: an increase in the number of cache misses due to VM switching, and an increase in the number of instructions or the data access latency caused by access to the same main memory unit. To solve these problems, we considered two approaches. Our solution to the first problem entailed introducing the rate at which the number of cache misses is increased when multiple VMs run, $MC_{mem\_dytpe\_mvm}$, which is obtained through an actual measurement [64]. The second solution is to introduce a queuing model modifying the method of Gulur and Govindarajan [65], who used M/D/1 [66] as the DRAM and a memory controller to estimate the latency.

Algorithm 1 presents the method for extending the join cost calculation to the cloud environment.

$$L_{mem} \text{ of prediction target CPU} = f(BW_{MM},\ L_{MM}), \tag{8.17}$$

where f is a function of a queuing model, BW_{MM} is bandwidth of main memory, and L_{MM} is main memory latency. In addition to this approach, a regression model is estimated using measured latency on a micro-benchmark. The micro-benchmark, such as STREAM benchmark [67], issues memory requests and controls the number of accessing arrays concurrently.

$$LC_{mem\_dt\_mvm} = \frac{L_{mem\_dt} \text{ of prediction target CPU}}{L_{mem\_dt} \text{ of reference CPU}} \tag{8.18}$$

Algorithm 1 Update cost calculation formula

```
for $x \in L1I, L1D, \cdots, LNI, LND, MMI, MMD$ do
    Calculate$M_{x\_hit\ or\ acc}$using $MC_x$and $MC_{x\_mvm}$
    in Eqn. (8.11)(8.12)(8.13)
    $L_x \leftarrow L_x \times LC_x \times LC_{x\_mvm}$
end for
Right-hand side of Eqn. (8.7) $\leftarrow$ Right-hand side of Eqn. (8.7) $\times BM$
Update$C_{join}$
```

8.3. Evaluation Method

Initially, we evaluated our proposed method by using a single physical server with a single VM running on it. We verified that the cost calculation formula using the statistical values measured on the reference CPU can accurately estimate the cross point of the join methods on CPUs of different generations. We performed the evaluation by using the database of the TPC-H benchmark and estimated the cross points between NLJ and HJ. The queries for measuring CPU statistical information through the performance monitor of the reference CPU comprise a two-table join and three-table join based on Q3 of TPC-H. The method we used to formulate the join cost conforms to the method proposed in Chapter 7. The combinations of TPC-H tables to join are as follows: Customer–Orders (C–O), Part–Lineitem (P–L), Supplier–Lineitem (S–L), Customer–Orders–Lineitem (C–O–L), Part–Lineitem–Supplier–Orders–Customer (P–L–S–O–C), and Supplier–Lineitem–Part–Orders–Customer (S–L–P–O–C). The queries to join are shown in Chapter 7. The CPU statistical information for formulating the cost calculation is measured through the reference CPU, the details of which are provided in Table 8.2. The prediction of the cross point of two join methods by using the cost calculation formula is aimed at determining case SF100 on the reference CPU and target CPU, as shown in Table 8.2. The reference CPU is a Westmere-based processor, whereas the target CPU is based on a Skylake-based processor, which is a CPU from a different generation. We refer to Westmere-based processors as Westmere and Skylake-based processors as Skylake in this paper. For further verification, we replaced the target CPU and used a small database to measure the statistical information because we aimed to minimize the cost of the cloud service operation. The cost calculation formula of Westmere is shown in Chapter 7. The cost calculation formula of Skylake was generated in this chapter. As we used a disk-based DBMS rather than an in-memory database, the join cost is the sum of the CPU and I/O costs. The formula to determine the I/O cost was estimated using the measured value determined in Chapter 7.

To apply the cost formula to the target CPU, MC_{Li_dtype}, LC_{mem_dt}, and CRL_{MP} were determined. For estimating Skylake's cost calculation formula by changing Westmere's cost calculation formula, the difference between the reference and target CPUs is calculated such that the associativity of the L1I cache is increased from 4-way to 8-way, the associativity of

Table 8.2.: Evaluation Environment

CPU (Westmere)	
CPU	Xeon L5630 2.13 GHz 4-core, LLC 12 MB [Westmere-EP] ×2
Memory	DDR3 12 GB (4 GB ×3) physically attached to only one CPU
Disk for DB	PCIe NVMe Flash SSD 800 GB
OS/DBMS	CentOS 6.6 (×64)/MariaDB 10.1.8 with InnoDB
DB size	TPC-H SF5 (5 GB) for generation of cost calculation formula and SF100 (100 GB) for validation
Measuring tool	[CPU events] VTune Amplifier, [Query, I/O] System Tap
CPU (Skylake)	
CPU	Xeon E3-1230 v5 3.4 GHz 4-core, LLC 8 MB [Skylake] ×1
Memory	DDR4 32 GB
Disk for DB	PCIe NVMe Flash SSD 800 GB
OS/DBMS	CentOS 6.6 (×64)/MariaDB 10.1.8 with InnoDB
DB size	TPC-H SF5 (5GB) for generation of cost calc. formula and SF100 (100 GB) for validation
Measuring tool	[CPU events] VTune Amplifier, [Query, I/O] System Tap

the L2 cache is decreased from 8-way to 4-way, and the capacity of L3, known as the last level cache (LLC), is reduced from 12 to 8 MB. Hill et al. [60] demonstrated that the instruction cache miss rate is decreased by approximately 0.06 from 4-way to 8-way ($MC_{L1\_I} = 1/1.06$) and the unified cache miss rate is increased by approximately 0.06 from 8-way to 4-way ($MC_{L2\_I\text{or}D} = 1.06$). In general, the cache miss rate is decreased as a power law of the cache size [62]. When this rule is applied, the cache miss rate is reduced by approximately 0.8 times from 12 MB to 8 MB ($MC_{L3\_I\text{or}D} = 1/\sqrt{8/12}$=1/0.8). CRL_{MP} = 1.1, which is calculated using the measured branch misprediction penalty according to the 7-Zip LZMA Benchmark [55]. However, when estimating Westmere's cost calculation formula by changing Skylake's cost calculation formula, all the above-mentioned parameters are used as reciprocals of cache miss ratio changes and branch misprediction penalty changes.

For the server with Westmere, we used a dual CPU, and for the Skylake server, we used a single CPU (Table 8.2). Therefore, the cost calculation model of Skylake included only the local main memory. When modifying the Skylake cost formula to Westmere, the Skylake cost formula used the average latency of local main memory, remote main memory, and remote L3 cache instead of local main memory latency.

8.4. Results and Discussion

We evaluated the proposed cost calculation methods by comparing the measured cross points and cross-point estimates using the modified cost calculation formula in Tables 8.3 and 8.4. The base cost calculation formula is created using the value measured on the reference CPU (row #1 of Tables 8.3 and 8.4). The effects of the changes in CPU frequency, memory latency, cache miss ratio, and the branch misprediction penalty in this order to the cost formula are reflected by the cross points shown from rows #2 to #5. The cross point measured on the target CPU is shown in row #6. The comparison results of the cross points in row #9 of Table 8.3

show that eight of the 12 join cases improved the accuracy through parameter modification. Negative values in row #9 imply that the estimation of cross-point accuracy deteriorated more than before the modification.

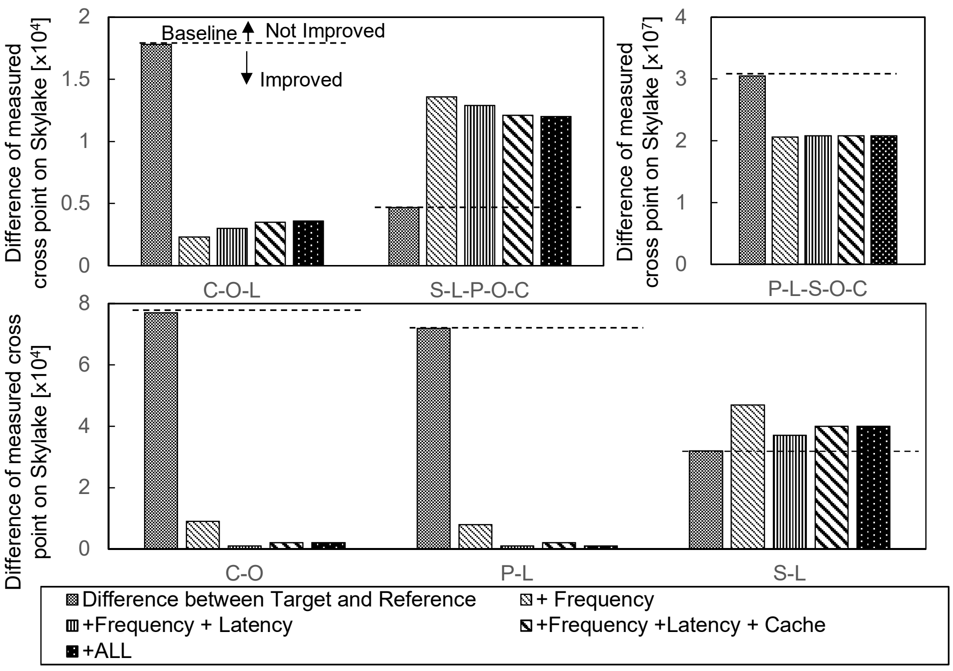


Figure 8.2.: Difference in Updating Architectural Changes for the Skylake Cost Model Based on Measurement Values of Westmere

The results show that, from among the modified coefficients (CPU frequency, latency, cache architecture, and branch misprediction penalty), CPU frequency is the most effective parameter (Figure 8.2 and Figure 8.3). To negate the effect of changing each CPU parameter, we analyzed the join cost and investigated the proportion of each change in Figure 8.4. The effect of changing other parameters was less than the effect of changing CPU frequency because

Table 8.3.: Accuracy of Estimating the Selectivity of the Cross Point of a Nested Loop Join and Hash Join when Applying the Westmere-based Model to Skylake

# Join tables	C–O	P–L	S–L	C–O–L	S-L-P-O-C	P-L-S-O-C
1 [Reference] Model of Westmere	2.89×10^3	2.72×10^3	2.64×10^3	8.08×10^4	9.27×10^4	5.15×10^7
2 Modifying Frequency	2.03×10^3	1.92×10^3	1.85×10^3	6.53×10^4	7.44×10^4	4.16×10^7
3 Modifying Latency +#2	2.13×10^3	2.01×10^3	1.95×10^3	6.60×10^4	7.51×10^4	4.18×10^7
4 Modifying Cache +#3	2.10×10^3	1.98×10^3	1.92×10^3	6.65×10^4	7.59×10^4	4.18×10^7
5 Modifying Branch. +#4	2.10×10^3	1.99×10^3	1.92×10^3	6.66×10^4	7.60×10^4	4.18×10^7
6 [Target] Measured on Skylake	2.12×10^3	2.00×10^3	2.32×10^3	6.30×10^4	8.80×10^4	2.10×10^7
7 |#1 – #6|	7.70×10^4	7.20×10^4	3.20×10^4	1.78×10^4	4.70×10^5	3.05×10^7
8 |#5 – #6|	2.00×10^5	1.00×10^5	4.00×10^4	3.60×10^5	1.20×10^4	2.08×10^7
9 Improvement |#7 – #8|	7.50×10^4	7.10×10^4	-8.00×10^5	1.42×10^4	-7.30×10^5	9.70×10^8

[Note] C: Customer, O: Orders, L: Line item, P: Part, S: Supplier, Branch.: Branch misprediction

Table 8.4.: Accuracy of Estimating the Selectivity of the Cross Point of a Nested Loop Join and Hash Join when Applying the Skylake-based Model to Westmere

# Join tables	C–O	P–L	S–L	C–O–L	S-L-P-O-C	P-L-S-O-C
1 Reference Model of Skylake	2.16×10^3	2.04×10^3	1.86×10^3	6.61×10^4	7.28×10^4	3.90×10^7
2 Modifying Freqency	3.22×10^3	3.02×10^3	2.68×10^3	8.64×10^4	9.63×10^4	5.22×10^7
3 Modifying Latency +#2	3.02×10^3	2.83×10^3	2.55×10^3	8.44×10^4	9.42×10^4	5.09×10^7
4 Modifying Cache +#3	3.00×10^3	2.81×10^3	2.53×10^3	8.38×10^4	9.36×10^4	5.06×10^7
5 Modifying Branch. +#4	3.00×10^3	2.81×10^3	2.53×10^3	8.38×10^4	9.35×10^4	5.06×10^7
6 [Target] Measured on Westmere	2.77×10^3	2.40×10^3	2.54×10^3	8.96×10^4	1.22×10^3	2.90×10^7
7 |#1 – #6|	6.10×10^4	3.60×10^4	6.80×10^4	2.35×10^4	4.92×10^4	1.00×10^7
8 |#5 – #6|	2.30×10^4	4.10×10^4	1.00×10^5	5.80×10^5	2.85×10^4	2.16×10^7
9 Improvement |#7 – #8|	3.80×10^4	-5.00×10^5	6.70×10^4	1.77×10^4	2.07×10^4	-1.16×10^7

[Note] C: CUSTOMER, O: ORDERS, L: LINEITEM, P: PART, S: SUPPLIER, Branch.: Branch misprediction

the impact on the performance parameters resulting from the architectural change was less than the change in CPU frequency. All the components, such as cache memory access, main memory access, branch misprediction penalty, and I/O are functions of selectivity in Chapter 7. Each component of Figure 8.4 was changed because the cross point was changed in each case. However, the size of each component was different, and a larger component had a greater effect on cost value. In the NLJ case, frequency, I/O, and L1 Data were effective. However, these parameters were not included in the Westmere and Skylake differences, and only frequency was an effective parameter in the NLJ case.

In the case of the HJ build phase, the costs of *L2 Instruction* and *L1 Data* were greater than other costs. In the case of the HJ probe phase, the costs of *L2 instruction*, *L1 Data*, and *I/O* were greater than other costs. The cost of *I/O* depends on the selectivity of the cross point. Therefore, changes to L2 cache and CPU frequency were dominant for the join cost. In addition, Figures 8.2 and 8.3 show that our cost calculation formula should be updated when CPU frequency or L1 cache architecture are updated.

Incidentally, in the HJ build phase, join cost did not include I/O cost because the main thread of MariaDB-measured CPU events does not issue many disk I/O requests, while its I/O threads issued a lot of asynchronous disk I/O requests in our configuration. In the HJ probe phase of the three-table join, MariaDB accesses the third table the same as in NLJ and issues a synchronous disk I/O. Hence, the cost of the probe phase includes the I/O cost (Figures 8.2 and 8.3).

When modifying the Westmere cost formula to Skylake, the accuracy of estimating the cross point of the S-L join and S-L-P-O-C join was lower than other join cases. Each modified cost model of NLJ or HJ approached the target cost measured on Skylake in Figure 8.5(a) and (b). However, the cross point estimated by the modified cost formula moved further away from the target cross point of the unmodified cost formula. In the case of the cross point of the P-L join and P-L-S-O-C join using the modified Skylake cost formula to Westmere, the same problem also occurs, as shown in Figure 8.5(c) and (d). From the above, although our proposed method of modifying measurement-based cost formulas are effective, it does not always improve the accuracy of predicted cross points. To further improve the prediction accuracy of the cross point, it is necessary to study a method for directly controlling the cross points.

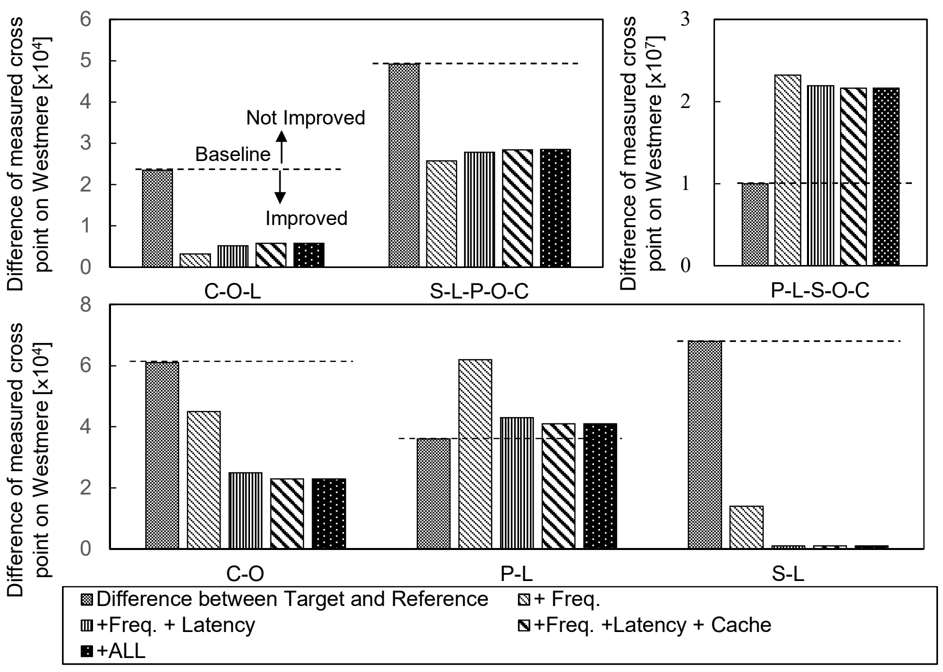


Figure 8.3.: Difference in Updating Architectural Changes for the Westmere Cost Model Based on Measurement Values of Skylake

Considering the limits of our proposed method, our join cost calculation method cannot overcome large CPU updates, such as pipeline structure level updates, which would require the cost calculation formula to be prepared in advance for each CPU.

8.5. Conclusion

Performance changes, such as memory latency and cache size, introduced because of architectural changes using different generations of CPUs were reflected in the measurement-based cost calculation formula for join processing. By using this updated cost calculation formula, we verified that it is possible to select the join method of NLJ and HJ accurately when considering CPUs of different generations. As a result, it was determined that the cross point was estimated with an accuracy of 66% for the join test cases analyzed. In conclusion, we verified that our proposed method for the measurement-based join cost calculation can be utilized to calculate the cost of a join operation for CPUs of different generations, thereby contributing to a reduction in the operational costs of a cloud service platform.

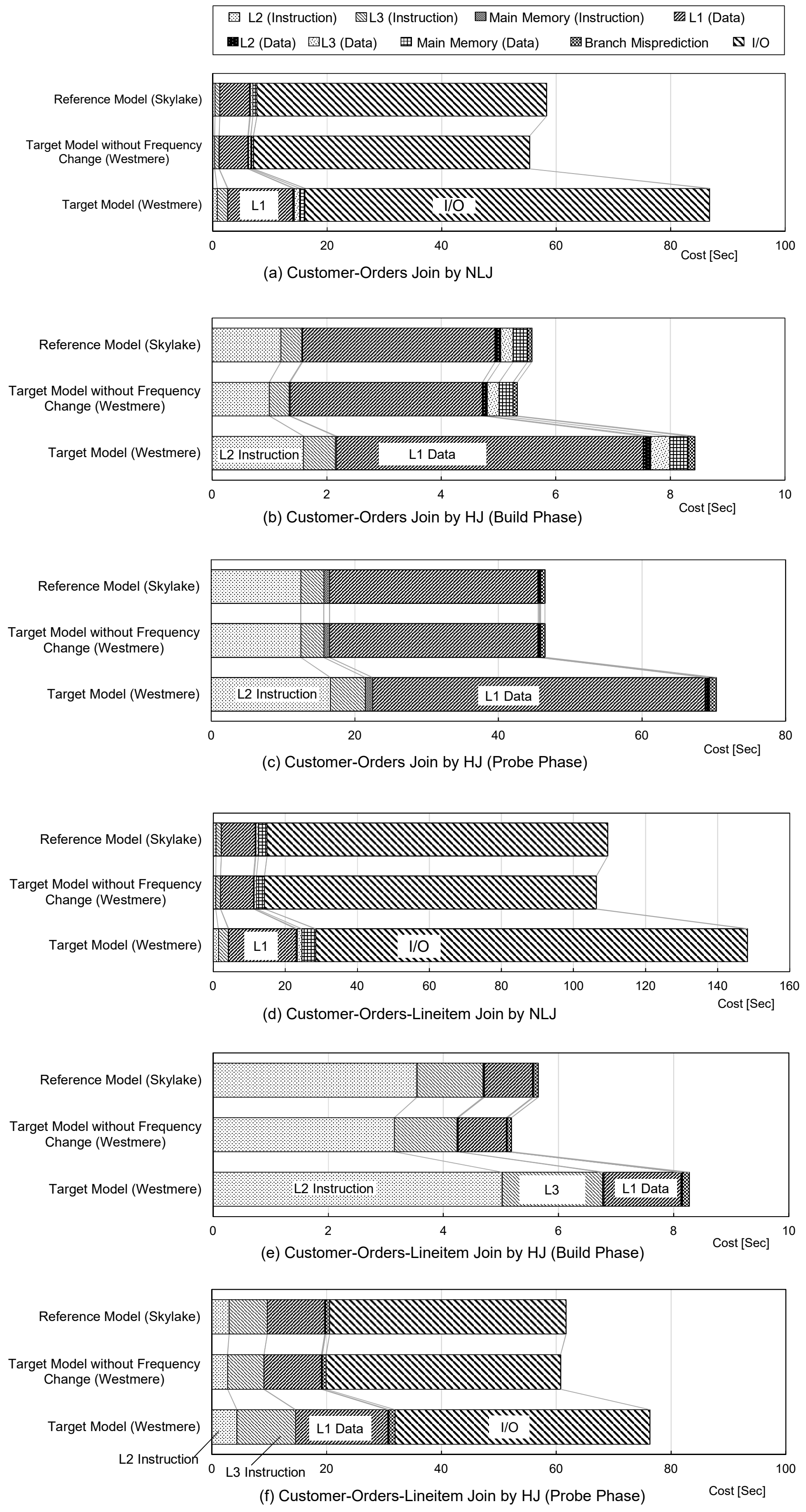


Figure 8.4.: Decomposition of the Join Costs of the C–O and C–O–L Tables

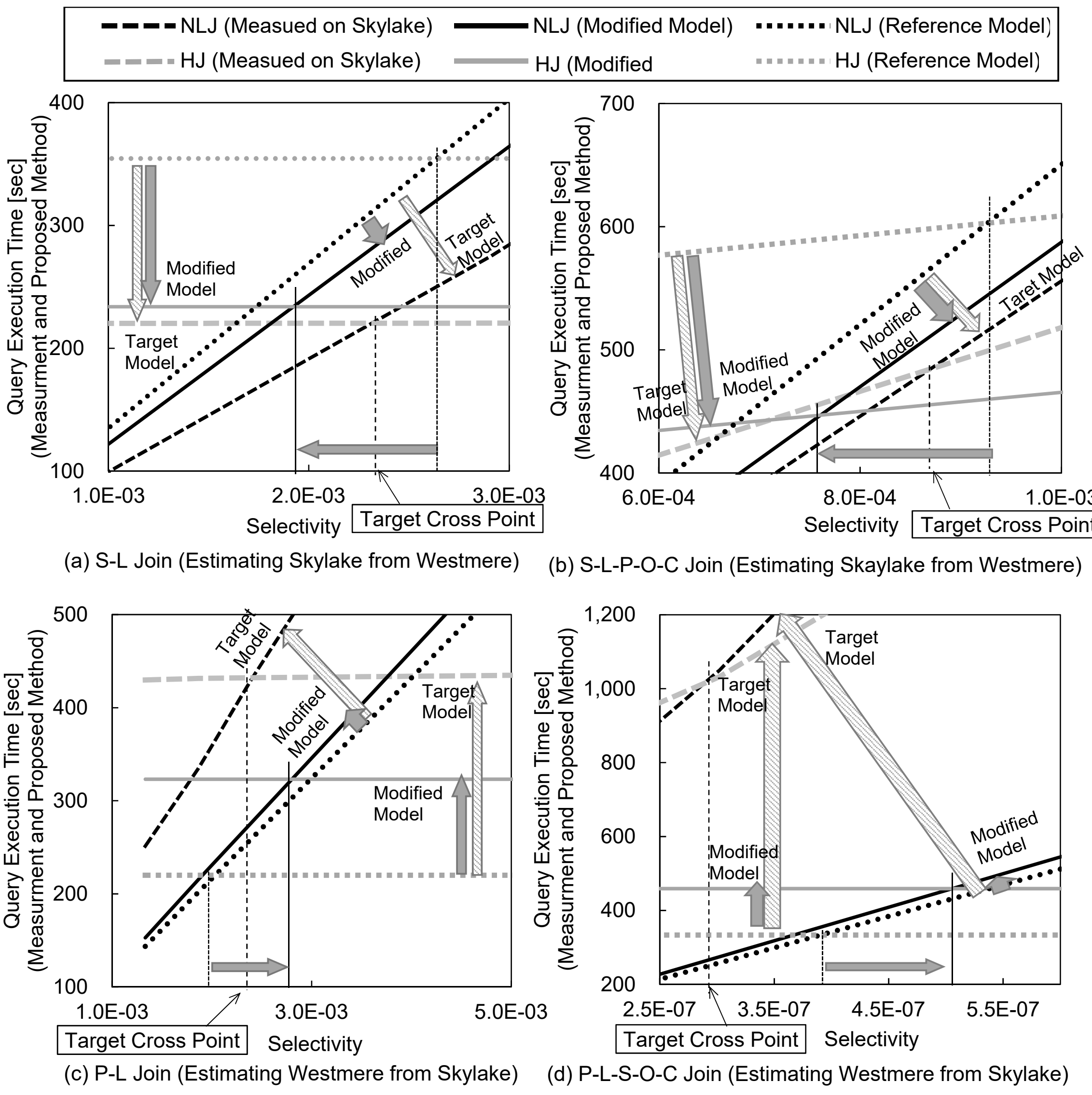


Figure 8.5.: Reason for Lower Accuracy of Estimated Cross Points after Model Modification

9. Conclusions and Future Works

To improve accuracy of join cost calculation for database query optimization, we proposed the measurement-based cost calculation method. We focused on CPU pipeline architecture to estimate accurate elapsed time of analysis queries as CPU of analysis queries. The CPU cost is composed of three elements, data cache access time, instruction cache miss penalty and branch misprediction recovering penalty. Each cost elements are given by regression analysis of measured CPU statistical information such as number of cache hits during executing a join operation. For supporting multiple table join, repeating operations are found from a access path of the query optimizer generated of DBMS and are modeled. First, we created the proposed cost calculation using 100GB TPC-H database when applying two-tables join. The join operations were NLJ and HJ. We evaluated the accuracy of intersection of NLJ cost formula and HJ cost formula. As a result, the difference between the predicted cross point and the measured cross point was less than 0.1% selectivity and was reduced by 71% to 94% compared with the difference between the cross point obtained by the conventional method and the measured cross point. Second, we modified the proposed two-tables join cost calculation method to support multi-tables join. In addition, to reduce database administrator's operation cost, we used the cost calculation formulas created using small size, 5GB database and reduced measurement time. The experimental results showed the ratio of accuracy improvement was 74% to 95%. Finally, we evaluated our proposed method could apply different generation CPUs with small change of cost calculation formula's coefficients. We utilized other research results such as change rate of cache miss ratio against changes in associativity or size to modify the coefficients. We verified that it is possible to select the join method of NLJ and HJ accurately when considering CPUs of different generations. As a result, it was determined that the cross point was estimated with an accuracy of 66% for the join test cases analyzed. In conclusion, 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. Our proposed method for the measurement-based join cost calculation can be utilized to calculate the cost of a join operation for CPUs of different generations, thereby contributing to a reduction in the operational costs of a cloud service platform.

In the future, 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. Moreover, to support cloud environment more, we will propose how to modify the measurement-based cost calculation method for single database instance for multiple database instance running environment.

Furthermore, not only effort of improvement accuracy of cost estimation but also improvement accuracy of estimating distribution of attributes of data are required to improve accuracy for multi-table join. Therefore, we will tackle develop a method of estimating data distribution suitable for the proposed cost calculation method.

References

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

[2] Andy 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].

[3] Zora Caklovic, Product Expert, Oliver Rebholz, et al. Bringing persistent memory technology to sap hana: Opportunities and challenges. *Annual SNIA Persistent Memory Summit*, 2017.

[4] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. Memory management techniques for large-scale persistent-main-memory systems. *Proceedings of the VLDB Endowment*, 10(11):1166–1177, 2017.

[5] Transaction Processing Performance Council (TPC). TPC BENCHMARK™ H (decision support) standard specification revision 2.17.3. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.3.pdf, January 2018.

[6] Andy Rudoff. Persistent memory programming. *Login: The Usenix Magazine*, 42:34–40, 2017.

[7] Yannis Ioannidis. The history of histograms (abridged). In *Proceedings 2003 VLDB Conference*, pages 19–30. Morgan Kaufmann, San Francisco, 2003.

[8] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, November 2015.

[9] P. Griffiths 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*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM.

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

[11] Michael V. Mannino, Paicheng Chu, and Thomas Sager. Statistical profile estimation in database systems. *ACM Comput. Surv.*, 20(3):191–221, September 1988.

[12] Yannis E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, March 1996.

[13] Viswanath Poosala, Peter J. Haas, Yannis E. Ioannidis, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. *SIGMOD Rec.*, 25(2):294–305, June 1996.

[14] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 256–276, New York, NY, USA, 1984. ACM.

[15] Oystein Grovlen. Histogram Support in Mysql 8.0. https://www.slideshare.net/oysteing/histogram-support-in-mysql-80 [retrieved: Jan, 2019], February 2018.

[16] MariaDB Knowlegde Base – Histogram-Based Statistics. https://mariadb.com/kb/en/library/histogram-based-statistics/ [retrieved: Jan, 2019].

[17] Viswanath Poosala and Yannis E Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB*, volume 97, pages 486–495, 1997.

[18] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. Random sampling for histogram construction: How much is enough? *SIGMOD Rec.*, 27(2):436–447, June 1998.

[19] Yannis E. Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, SIGMOD '91, pages 268–277, New York, NY, USA, 1991. ACM.

[20] Weimin Du, Ravi Krishnamurthy, and Ming-Chien Shan. Query optimization in a heterogeneous dbms. In *VLDB*, volume 92, pages 277–291, 1992.

[21] Qiang Zhu and Per-Åke Larson. Building regression cost models for multidatabase systems. In *Proceedings of the Fourth International Conference on on Parallel and Distributed Information Systems*, DIS '96, pages 220–231, Washington, DC, USA, 1996. IEEE Computer Society.

[22] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[23] Olav Sandstå. Mysql Cost Model. http://www.slideshare.net/olavsa/mysql-optimizer-cost-model [retrieved: Jan, 2018], October 2014.

[24] Stefan Manegold, Peter A Boncz, and Martin L Kersten. Optimizing database architecture for the new bottleneck: memory access. *The VLDB Journal*, 9(3):231–246, 2000.

[25] Stefan Manegold, Peter Boncz, and Martin L Kersten. Generic database cost models for hierarchical memory systems. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 191–202. VLDB Endowment, 2002.

[26] Michael S. Kester, Manos Athanassoulis, and Stratos 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*, SIGMOD '17, pages 715–730, New York, NY, USA, 2017. ACM.

[27] Amira Rahal, Qiang Zhu, and Per-Åke Larson. Evolutionary techniques for updating query cost models in a dynamic multidatabase environment. *The VLDB Journal*, 13(2):162–176, May 2004.

[28] Qiang Zhu, S. Motheramgari, and Yu 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*, DEXA '00, pages 509–525, London, UK, UK, 2000. Springer-Verlag.

[29] Mumtaz Ahmad, Songyun Duan, Ashraf Aboulnaga, and Shivnath Babu. Predicting completion times of batch query workloads using interaction-aware models and simulation. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 449–460, New York, NY, USA, 2011. ACM.

[30] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and David 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*, number DIAS-CONF-1999-001, pages 266–277, 1999.

[31] Richard A. Hankins, Trung Diep, Murali Annavaram, Brian Hirano, Harald Eri, Hubert Nueckel, and John P. Shen. Scaling and charact rizing database workloads: Bridging the gap between research and practice. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 151–, Washington, DC, USA, 2003. IEEE Computer Society.

[32] Tsuyoshi Tanaka, Toshiaki Tarui, and Ken Naono. Investigating suitability for server virtualization using business application benchmarks. In *Proceedings of the 3rd international workshop on Virtualization technologies in distributed computing*, pages 43–50. ACM, 2009.

[33] Dileep Bhandarkar and Jason Ding. Performance characterization of the pentium pro processor. In *High-Performance Computer Architecture, 1997., Third International Symposium on*, pages 288–297. IEEE, 1997.

[34] Kimberly Keeton, David A Patterson, Yong Qiang He, Roger C Raphael, and Walter E Baker. Performance characterization of a quad pentium pro smp using oltp workloads. In *ACM SIGARCH Computer Architecture News*, volume 26, pages 15–26. IEEE Computer Society, 1998.

[35] R. Murphy. On the effects of memory latency and bandwidth on supercomputer application performance. In *2007 IEEE 10th International Symposium on Workload Characterization(IISWC)*, volume 00, pages 35–43, 09 2007.

[36] Stijn Eyerman and Lieven Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE micro*, 28(3), 2008.

[37] Richard E Matick, Thomas J Heller, and Michael Ignatowski. Analytical analysis of finite cache penalty and cycles per instruction of a multiprocessor memory hierarchy using miss rates and queuing theory. *IBM Journal Of Research And Development*, 45(6):819–842, 2001.

[38] Richard E Matick. Comparison of analytic performance models using closed mean-value analysis versus open-queuing theory for estimating cycles per instruction of memory hierarchies. *IBM Journal of Research and Development*, 47(4):495–517, 2003.

[39] Intel® 64 and ia-32 architectures software developer's manual, volumes 3a, 3b, 3c, and 3d: System programming guide, 2018.

[40] Peter A Boncz, Stefan Manegold, Martin L Kersten, et al. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, volume 99, pages 54–65, 1999.

[41] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A performance counter architecture for computing accurate cpi components. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 175–184, New York, NY, USA, 2006. ACM.

[42] Omesh Tickoo, Ravi Iyer, Ramesh Illikkal, and Don Newell. Modeling virtual machine performance: Challenges and approaches. *SIGMETRICS Perform. Eval. Rev.*, 37(3):55–60, January 2010.

[43] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 379–391, New York, NY, USA, 2013. ACM.

[44] David Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. *Intel Performance Analysis Guide*, 30:18, 2009.

[45] Padma Apparao, Ravi Iyer, and Don Newell. Towards modeling & analysis of consolidated CMP servers. *SIGARCH Comput. Archit. News*, 36(2):38–45, May 2008.

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

[47] P. Trancoso, J. . Larriba-Pey, Z. Zhang, and J. Torrellas. The memory performance of dss commercial workloads in shared-memory multiprocessors. In *Proceedings Third International Symposium on High-Performance Computer Architecture*, pages 250–260, Feb 1997.

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

[49] A. Patel, F. Afram, S. Chen, and K. Ghose. Marss: A full system simulator for multicore x86 cpus. In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1050–1055, June 2011.

[50] J. L. Lo et al. An analysis of database workload performance on simultaneous multithreaded processors. In *ACM SIGARCH Computer Architecture News*, volume 26, pages 39–50. IEEE Computer Society, 1998.

[51] The MariaDB foundation - ensuring continuity and open collaboration in the mariadb ecosystem, 2017.

[52] 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*, pages 261–270, Sept 2009.

[53] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems: the complete book, Second Edition*, pages 804–807. Pearson Education Limited, 2014.

[54] Annita N. Wilschut, Jan Flokstra, and Peter M. G. Apers. Parallel evaluation of multi-join queries. *SIGMOD Rec.*, 24(2):115–126, May 1995.

[55] Igor Pavlov. 7-Zip LZMA Benchmark. http://www.7-cpu.com/ [retrieved: January, 2018], 2017.

[56] A. Aboulnaga et al. Automated statistics collection in DB2 UDB. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 1158–1169. VLDB Endowment, 2004.

[57] 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].

[58] Intel Corporation. Intel 64® and IA-32 architectures optimization reference manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, January 2018.

[59] Jack Doweck, Wen-Fu Kao, Allen Kuan yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro*, 37(2):52–62, Mar 2017.

[60] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38(12):1612–1630, Dec 1989.

[61] Rathijit Sen and Karthik Ramachandra. Characterizing resource sensitivity of database workloads, Feb. 2018. 24th IEEE International Symposium on High-Performance Computer Architecture (HPCA 2018).

[62] A. Hartstein, V. Srinivasan, T. R. Puzak, and P. G. Emma. On the nature of cache miss behavior: Is it $\sqrt{2}$?, June 2008.

[63] S. Eyerman, J. E. Smith, and L. Eeckhout. Characterizing the branch misprediction penalty. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 48–58, March 2006.

[64] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 13–23, New York, NY, USA, 2005. ACM.

[65] Nagendra Gulur, Mahesh Mehendale, and Ramaswamy Govindarajan. A comprehensive analytical performance model of dram caches. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 157–168, New York, NY, USA, 2015. ACM.

[66] Robert B. Cooper. *Introduction to Queueing Theory*. North Holland, 2nd edition, 1981.

[67] John D. McCalpin. Memory bandwidth: Stream benchmark performance results. http://www.cs.virginia.edu/stream/ [retrieved: March, 2018].

Related Publications

Journal papers

1. Tsuyoshi Tanaka and Hiroshi Ishikawa, CPU cost estimation method for relational database query execution to improve accuracy of join method selection. *The IEICE Transactions on Information and Systems (Japanese Edition)*, J100-D(4), pages 485–499, April 2017 (in Japanese).
2. Tsuyoshi Tanaka and Hiroshi Ishikawa, Measurement-based cost estimation method for multi-table join operation in an in-memory database, International Journal on Advances in Software, Vol. 10, No. 3&4: pages 459–476, 2017.

International Conference papers

1. Tsuyoshi Tanaka and Hiroshi Ishikawa, Measurement-based cost estimation method of a join operation for an in-memory database, In *The Ninth Internationa Conferences on Advances in Multimedia*, MMEDIA2017, pages 57–66, Venice, Italy, April 2017.
2. Tsuyoshi Tanaka and Hiroshi Ishikawa. Evaluation on applicability of measurement-based join cost calculation method using different generation CPUs. In *Proceedings IEEE 2018 International Congress on Cybermatics, The 18th IEEE Computer and Information Technology*, CIT-2018, pages 1894–1901. IEEE Computer Society, 2018.

Domestic Conference papers

1. Tsuyoshi Tanaka and Hiroshi Ishikawa, Cost estimation method based on CPU architecture for relational database query optimization, In *IEICE Technical Report*, vol. 115, no. 399, CPSY2015-116, pages 67–72, Jan. 2016 (in Japanese).
2. 田中剛, 石川博, ジョイン方式選択の精度向上をめざしたデータベース問い合わせ処理における CPU 処理コスト計算方法の検討とその評価, 第 8 回データ工学と情報マネジメントに関するフォーラム, DEIM 2016, 2016 年 3 月 (in Japanese).

Acknowledgments

First of all, I would like to express my deepest gratitude to my supervisor, Prof. Hiroshi Ishikawa from Department of Information and Communication Systems, Graduate School of System Design, Tokyo Metropolitan University for the continuous support of my doctoral study and related research, for insightful comments and suggestions. Without his guidance and persistent help this dissertation would not have been possible.

Next, I would like to express my gratitude to my examiner, Assoc. Prof. Shohei Yokoyama from Graduate School of System Design, Assoc. Prof. Kaoru Katayama from Graduate School of System Design and Prof. Jun Miyazaki from Department of Computer Science, School of Computing, Tokyo Institute of Technology. Thanks also to Assoc. Prof. Masaharu Hirota and Research Assistant Prof. Tetsuya Araki for constructive comments and warm encouragement. I would like to offer my special thanks to Emeritus Professor Yoshifumi Masunaga from Ochanomizu University and Dr. Yasuhiro Fujiwara from NTT Software Innovation Center for giving me great suggestions to the research policy and evaluation method at DEIM 2016.

Then, I would like to express my gratitude to Hitachi, Ltd. Research & Development Group for their financial support. I would also like to express my gratitude to the members of the Data Science Research Department of Research & Development Group, Hitachi, Ltd., for providing business support on the balance between daily work and studies. I had the support and encouragement of Assoc. Prof. Masaki Endo from Polytechnic University as senior student in the Ishikawa's laboratory. I was encouraged by discussing Mr. Daiju Kato, who entered the same year, about our jobs and studies. I received much cooperation from the members of Ishikawa's laboratory.

Finally, I would also like to express my gratitude to my wife, my daughter and other family members for their moral support and warm encouragements.

A. Appendix

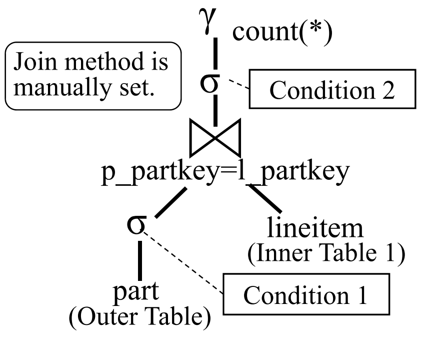
A.1. Queries for Evaluating Cost Calculation Formulas

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

(a) SQL

```
select count(*)
from  part, lineitem
where
   (p_type='STANDARD ANODIZED TIN'
   or p_type='STANDARD ANODIZED STEEL)     } Condition 1
   and p_size < N
   and p_partkey = l_partkey
   and l_shipdate < date '1995-03-06';     } Condition 2
```

(b) Access Path



(c) Selection Condition and Selectivity

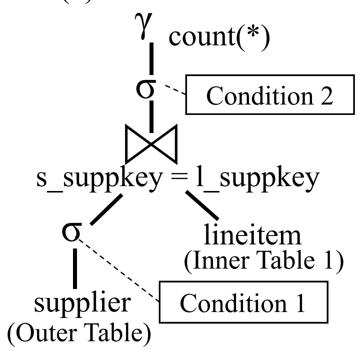
N	6	8	10	20	30	40	50
Selectivity P_O (Condition 1)	1.33×10^{-3}	1.87×10^{-3}	2.40×10^{-3}	5.07×10^{-3}	7.73×10^{-3}	1.04×10^{-2}	1.31×10^{-2}

Figure A.1.: Target Query of Cost Estimation for Part and Lineitem Join

(a) SQL

```
select count(*)
from supplier, lineitem                Condition 1
where
   s_acctbal > N                   }
   and s_nationkey = 0             }
   and s_suppkey = l_suppkey
   and l_shipdate > date '1995-03-06'; }  Condition 2
```

(b) Access Path



(c) Selection Condition and Selectivity

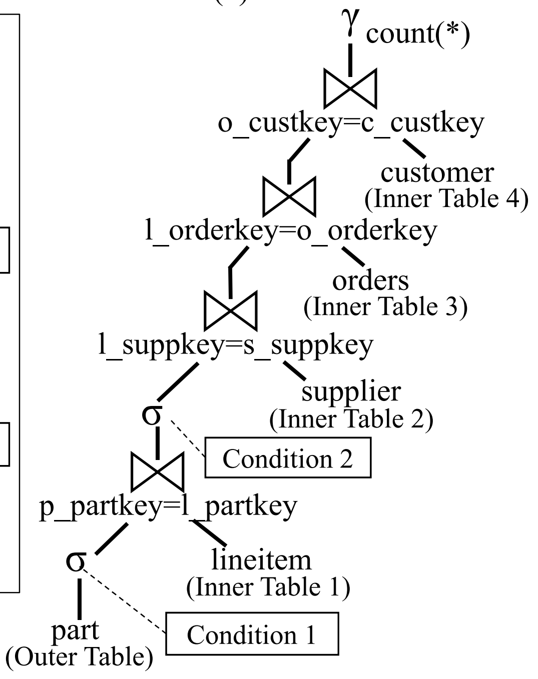
N	9998	9978	9798	9200	9000	8000	7000
Selectivity P_O (Condition 1)	7.24×10^{-6}	8.00×10^{-5}	7.35×10^{-4}	2.91×10^{-3}	3.64×10^{-3}	7.27×10^{-3}	1.09×10^{-2}

Figure A.2.: Target Query of Cost Estimation for Supplier and Lineitem Join

(a) SQL

```
select count(*)
from
   part
   STRAIGHT_JOIN lineitem
   STRAIGHT_JOIN supplier
   STRAIGHT_JOIN orders
   STRAIGHT_JOIN customer                 Condition 1
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' }  Condition 2
   and l_quantity = 10                }
   and l_orderkey = o_orderkey
   and o_custkey = o_custkey
   and l_suppkey = s_suppkey;
```

(b) Access Path



(c) Selection Condition and Selectivity

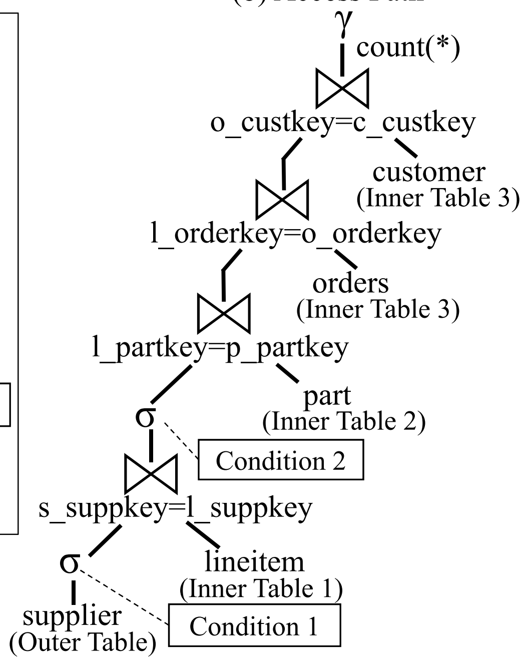
N	6	8	10	20	30	40	50
Selectivity P_O (Condition 1)	1.33×10^{-3}	1.87×10^{-3}	2.40×10^{-3}	5.07×10^{-3}	7.73×10^{-3}	1.04×10^{-2}	1.31×10^{-2}

Figure 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) Access Path



(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}

Figure A.4.: Target Query of Cost Estimation for Supplier, Lineitem, Part, Orders and Customer join

A.2. Measured CPU Counters

The list of CPU counters for modeling the CPU cost calculation are shown in Table A.1 and A.4. The constants and intermediate variable for modeling cost calculation formulas are shown in Table A.2 and A.5. The column of “Symbol” means variables in the cost calculation formulas. Intermediate Variables to obtain cost calculation formulas are shown in Table A.3 and A.4.

Table A.1.: Lists of CPU Counters to Measure and Preprocess of Westmere Processor

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

Table A.2.: Lists of Constants of Westmere Processor

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

Table A.3.: Lists of Intermediate Variable of Westmere Processor

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	$E33-E27-E69$

Table A.4.: Lists of CPU Counters to Measure and Preprocess of Skylake Processor

No.	Counter Name
$F1$	INST_RETIRED.ANY
$F2$	CPU_CLK_UNHALTED.THREAD
$F3$	BR_MISP_RETIRED.ALL_BRANCHES
$F4$	L2_RQSTS.CODE_RD_HIT
$F5$	L2_RQSTS.CODE_RD_MISS
$F6$	MEM_INST_RETIRED.ALL_LOADS_PS
$F7$	MEM_LOAD_RETIRED.FB_HIT_PS
$F8$	MEM_LOAD_RETIRED.L1_HIT_PS
$F9$	MEM_LOAD_RETIRED.L1_MISS_PS
$F10$	MEM_LOAD_RETIRED.L2_HIT_PS
$F11$	MEM_LOAD_RETIRED.L2_MISS_PS
$F12$	MEM_LOAD_RETIRED.L3_HIT_PS
$F13$	MEM_LOAD_RETIRED.L3_MISS_PS
$F14$	OFFCORE_RESPONSE:request=DEMAND_CODE_RD:response =L3_MISS_LOCAL_DRAM.ANY_SNOOP
$F15$	UOPS_EXECUTED.STALL_CYCLES

Table A.5.: Lists of Constants of DB Server with Skylake Processor

No.	Symbol	Events	Value
$F16$	–	CPU frequency [GHz] (Xeon E3–1250v5)	3.4
$F17$	L_{L1}	L1I Latency [cycle]	4
$F18$	L_{L1}	L1D Latency [cycle]	4
$F19$	L_{L2}	L2 Latency [cycle]	12
$F20$	L_{LLLC}	Local LLC Latency [cycle]	42
$F21$	L_{RLLC}	Remote LLC Latency [cycle]	—
$F22$	L_{LMM}	Local Main Memory Latency [cycle]	$F21 + 51 \times F19$
$F23$	—	Remote Main Memory Latency [cycle]	—
$F24$	L_{MP}	Branchmiss prediction cycle	16.5

Table A.6.: Lists of Intermediate Variable of Skylake Processor

No.	Symbol	Events	Calculation Formula for Preprocessing
$F25$	I_{Load}	LOAD instruction	$F6$
$F26$	M_{L1D}	L1D Hit (data)	$F25 \times (F8/(F7 + F8 + F9))$
$F27$	—	L1D Miss (data)	$F25 \times (F9/(F7 + F8 + F9))$
$F28$	M_{L2D}	L2 Hit (data)	$F27 \times (F10/(F10 + F11))$
$F29$	—	L2 Miss (data)	$F27 - F28$
$F30$	M_{LLLCD}	LLC Hit (data)	$F29 \times (F12/(F12 \times F13))$
$F31$	—	LLC Miss (data)	$F29 - F30$
$F32$	M_{RLLCD}	Remote LLC Hit (data)	—
$F33$	M_{LMMD}	Local Main Memory (data)	$F31$
$F34$	—	Remote Main Memory (data)	—
$F35$	—	Total Data Access Latency	$F26 \times F18 + F28 \times F19 + F30 \times F20 + F33 \times F22$
$F36$	I	Instruction	$F1 + F3$
$F37$	M_{L1I}	L1I Hit (instruction)	$F36 - F4 - F5$
$F38$	—	L1I Miss (instruction)	$F36 - F37$
$F39$	M_{L2I}	L2 Hit (instruction)	$F38 \times (F4/(F4 + F5))$
$F40$	—	L2 Miiss (instruction)	$F38 - F39$
$F41$	M_{LLLCI}	Local LLC Hit (instruction)	$F40 - F14$
$F42$	—	Local LLC Miss (instruction)	$F44$
$F43$	M_{RLLCD}	Remote LLC Hit (instruction)	—
$F44$	M_{LMMD}	Local Main Memory (instruction)	$F14$
$F45$	—	Remote Main Memory (instruction)	—
$F46$	—	Total Instruction Access Latency	$F39 \times F19 + F41 \times F20 + F44 \times F22$
$F47$	$C_{DCacheAcc}$	Data Access	$F2 - F15$
$F48$	C_{MP}	Branch Mispredition Penalty	$F3 \times F24$
$F49$	$C_{ICacheMiss}$	Instruction Penalty	$F15 - F48$