데이터 스트림에서 그래프 기반 기법을 이용한 슬라이딩
윈도우 다중 조인 처리

Processing Sliding Window Multi-Joins using a Graph-Based
Method over Data Streams

장 희*, Liang Zhang, 유흔섭**, Byeong-Seob You, 거준위***, Jun-Wei Ge,
김경배****, Gyoung-Bae Kim, 이순조***** Soon-Jo Lee, 배해영****** Hae-Young Bae

요 약
데이터 스트림 환경에서 셋 이상의 스트림들에 대한 조인연산을 위해 순서를 선택하는 기존 기법들은 항상 간단한 허리스틱 방법을 이용하였다. 그러나 기존 기법들은 조인 선택도나 데이터 수신 비율과 같은 것만 고려하여 일반적인 용용에서 비효율적이며 낮은 성능을 갖는다. 본 논문에서는 최적의 조인 순서로 그래프 기반의 슬라이딩 윈도우 다중 조인 알고리즘을 제안한다. 이 기법에서 슬라이딩 윈도우 조인 그래프를 먼저 생성하는데, 정점(vertex)은 조인 연산으로 표현되고 옆지(edge)는 슬라이딩 윈도우들 사이의 조인관계를 나타낸다. 그리고 정점 가중치(vertex weight)와 옆지 가중치(edge weight)는 각각의 조인의 비율과 조인 연산들의 상호관계를 표현한다. 이에 데이터 스트림은 빠른 처리를 해야 하므로 메모리 기반의 그래프 기법을 사용한다. 이를 이용하여 최대값만을 이용하여 조인 연산을 수행하는 MVP 알고리즘을 개선하고 이의 그래프에서 최적의 조인 순서를 찾는다. 이를 통한 최종 결과는 중첩-루프(nested loop) 조인 계획을 수행하여 얻어진다. 성능비교를 통하여 제안기법이 기존 기법보다 우수함을 증명한다.

Abstract
Existing approaches that select an order for the join of three or more data streams have always used the simple heuristics. For their disadvantage — only one factor is considered and that is join selectivity or arrival rate, these methods lead to poor performance and inefficiency in some applications. The graph-based sliding window multi-join algorithm with optimal join sequence is proposed in this paper. In this method, sliding window join graph is set up primarily, in which a vertex represents a join operator and an edge indicates the join relationship among sliding windows, also the vertex weight and the edge weight represent...
the cost of join and the reciprocity of join operators respectively. Then the optimal join order can be found in the graph by using improved MVP algorithm. The final result can be produced by executing the join plan with the nested loop join procedure. The advantages of our algorithm are proved by the performance comparison with existing join algorithms.

주요어: 데이터 스트림, 그래프 이론, 윈도우 조인, 핀리 최적화

Keyword: Data Stream, Graph Theory, Window Join, Query Optimization

1. 서론

Data stream [1] appears in many new applications recent few years. Data stream is fast, endless, continuous and real-time, which is different from traditional static relations stored in disk. Some representative applications include processing telephone call records [2], monitoring Internet traffic [3] and sensor data [4]. The system that can process data stream is called Data Stream Management System (DSMS). Each element in a data stream can be seemed as \((s, t)\), where \(s\) is the data element and \(t\) indicates the corresponding timestamp.

It is impossible to store entire data for processing, as data stream is infinite and physical memory is limited. Therefore, sliding window [1] may be used for preserving the new arrival data that are more significant. The basic problem about sliding window is inserting and expiring tuples. So inserting and expiring tuples leads to two different re-execution strategies. The eager re-evaluation strategy generates new results after each new tuple arrives. The lazy one re-executes the query periodically, which is a more practical solution. The query in DSMS is being executed continuously that is called continuous query. Generally speaking, each sliding window maps to one data stream. A sliding window can be shared by many continuous queries, and also one continuous query could inquiry many sliding windows. The research in this paper is about the join sequence of many sliding windows in one query with lazy re-execution strategy.

A simple heuristic based on one parameter is used for appointing the join sequence of data stream in the literature, such as evaluating the most selective predicate first [5]. However, a graph-based model for relation join is proposed in [6]. The model is the weighted direction join graph that includes many significant factors and can indicate all possible query plans (various execution orders). Then using the heuristic MVP algorithm an effective spanning tree is found in the graph model. Each spanning tree is corresponding to a query plan.

As sliding window is a stream-to-relation operator [1], a correlative graph model can also be defined to representing sliding window join. In this graph a node represents a join operation and a directed edge indicates the existence of common sliding window between two nodes. Weight values with optimization required parameters including arrival rate, window size and join selectivity factor are imposed to each node and edge. The model can represent different kinds of sliding window join queries. Then we improve
the MVP algorithm according to the attribute of data stream to acquire a spanning tree with low cost. As the computing of execution plan is not exploited in [6], the nested loop join algorithm is used to execute the query plan. Finally, the whole procedure to process sliding window join is completed.

Several contributions have been made in this paper. First, we define a novel sliding window join graph and identify its convenience of analyzing join query. Then based on the sliding window join graph we develop a new integrated join algorithm including the step that arrange sliding windows in an optimal join order. Finally we describe our implementation of the proposed algorithm and present results from a detailed performance study of the implementations.

Table 1 lists the symbols used in this paper and their meanings. The rest of the paper is organized as follows: Section 2 describes related work and section 3 shows how to use graph-based approach to process the window join. Section 4 presents experimental study of the algorithm. Finally, Section 5 gives the conclusion.

### 2. Related works

The related works are classified into two parts: stream join; the traditional graph model and MVP algorithm.

#### 2.1 Stream join

Most recent works on joining processing over data streams are based on Symmetric Hash Join (SHJ) [7] which adopts the pipelining algorithm [8]. SHJ is extended to XJoin [9] by processing spill overflowing inputs to disk effectively when memory fills up. Data stream multi-joins operation also extends SHJ to two classifications that are those processing a series of pipelining binary joins [9] [10], and those defining a single, symmetric, multiway join operator, such as Mjoin [5], SteMs [11]. The proposed graph-based algorithm is related to appoint the sequence of many binary join operators. [10] not only proposes incremental multiway nested loop join and hash join algorithms, also presents a sensible join ordering heuristic that is sorting the join first.
whose join selectivity is small and assembling fast streams at or near the top of the query plan. [5] also researches the probing sequence of data streams in experiment. But all above papers do not discuss the join order of sliding window. Also no existing join algorithm includes the step of selecting join order. These problems are researched in this paper.

2.2 Graph model and MVP algorithm

[2] defines the weighted directed join graph (WDJG). In this graph, each node represents a join operation. If a common relation exists between two nodes, the edge is a physical edge (PE); if not, the edge is a virtual edge (VE). The direction of edge implies the join order. Edge weights indicate the impact of a join to the cost of the next join. E.g. if there are two join operators $\theta_j = \text{RBS}$ and $\theta_i = \text{ST}$, the edge weight of $\overrightarrow{v_jv_i}$ is shown as:

$$
\overline{w}_{ji} = \left\{ \begin{array}{ll}
\frac{|R||S| \times JCF_i}{|S|} & \text{if } \overrightarrow{v_jv_i} \text{ is PE} \\
\frac{|R||S| \times JCF_j}{|S|} & \text{if } \overrightarrow{v_jv_i} \text{ is VE}
\end{array} \right.
$$

(1)

Vertex weights represent the accumulated impact of a join sequence to a next join operator. The weight of $\theta_j$ is shown as $\bar{w}_i$ is the next join of $\theta_i$ and $\overrightarrow{v_jv_i}$ is PE:

$$
\bar{w}_i = \left( w_i^1, w_i^2 \right)
= \left( \prod_{j} \left( \frac{1}{|R|} \times w_j^1 \times \sum_{\theta_j} \left( \frac{w_j^2 - 1}{w_{ji}} \right) \times JCF_i \right) \right)
$$

(2)

A spanning tree becomes an execution plan in WDJG. For finding query plans faster, effective spanning tree (EST) is defined. An effective query plan can be found in shorter processing time by deleting ineffective spanning tree and reducing giant search space.

The Maximum Value Precedence (MVP) algorithm is a relatively low complexity and yet high efficiency algorithm for optimizing in WDJG. This algorithm finds a near optimal solution using only $O(n^2)$ time, lower than existing algorithm. Its idea is to reduce the cost of expensive join operations as early as possible [2]. Thus two steps are included in the algorithm. The first step is to choose an edge to reduce the costly vertex weight. The second step is to select an edge that causes the edges minimum increase to the result of joins.

3. Design of graph-based sliding window multi-join

In this chapter, we first introduce the sliding window join graph and also analyze the detailed computing of the weight in this model. We then propose the sliding window join algorithm based on this model.

3.1 Sliding window join graph

A Sliding Window Weighted Directed Join Graph (SWWDJG) is a weighted complete graph. Each vertex represents a join operation of two sliding windows. Each vertex is connected to another vertex via two edges in opposite directions. Physical edges (PE) and virtual edges (VE) are two kinds of edges in
each SWWDJG. Physical edges $\overline{v_{ij_i}}$ and $\overline{v_{ij_j}}$ exist between vertexes $v_i$ and $v_j$ only if there is a common sliding window between them, otherwise virtual edges $\overline{v_{ij_i}}$ and $\overline{v_{ij_j}}$ exists. The direction of an edge indicates an execution order of $\theta_i$ after $\theta_j$. Each vertex $v_j$ and edge $\overline{v_{ij}}$ is associated with a weight, $\overline{w_{ij}}$ and $\overline{w_{ji}}$ respectively. They are defined as (3) (4).

$$\overline{w_{ji}} = \begin{cases} &\left(\overline{w_{ji}^1}, \overline{w_{ji}^2}\right) \\ &\left(\overline{w_{ji}^1 \times SF_j}, \overline{w_{ji}^1 \times SF_j \times JCF}\right) \quad \overline{v_{ij_i}} \text{ is a PE} \\ &\left(1,1\right) \quad \overline{v_{ij_i}} \text{ is a VE} \end{cases}$$

(3)

$$\overline{w_{ij_j}} = \text{cost}(\theta_j)$$

(4)

SWWDJG is rather different from WDJG. Firstly, the join operation represented by node is processed between sliding windows instead of static relations. Secondly, when calculating the edge weight— the cardinality and the tuple size of relation are replaced by the window size and the element size in data stream. And lastly, vertex weight in WDJG is dependent on its preceding vertex weights, which makes it difficult to specify. Thus the vertex weight in SWWDJG is set to the cost of each join operator, which makes us compute the join cost easier.

### 3.2 Computing edge weight and vertex weight

As the difference of two kinds of sliding windows [1]: time-based sliding window and count-based sliding window is the window size, we only researched the detailed computation of edge weight and vertex weight about time-based sliding window. The edge weight and vertex weight of count-based sliding window is similar to its. For each kind of weight, a computation formula is proposed.

#### 3.2.1 Edge weight

For a virtual edge, the join operations of VE will not influence each other on the join cost. Hence, the weight of a VE is defined as $<1,1>$ to show that the joins are independent.

Now consider the PE. The tuple size of a sliding window is constant written as $|v_j|^* M_j$, $|w_j|^* M_j$. The window size is represented as $|v_j|^* JF_j$, $|w_j|^* JF_j$. As the join selectivity factor and the join concatenation factor can be normally collected from statistical information, the edge weight can be represented by above factors. So (3) can be reshown as:

$$\overline{w_{ji}} = \begin{cases} &\left(\overline{w_{ji}^1}, \overline{w_{ji}^2}\right) \\ &\left(\overline{w_{ji}^1 \times SF_j}, \overline{w_{ji}^1 \times SF_j \times JCF}\right) \quad \overline{v_{ij_i}} \text{ is a PE} \\ &\left(1,1\right) \quad \overline{v_{ij_i}} \text{ is a VE} \end{cases}$$

(5)

#### 3.2.1 Vertex weight

Vertex weight only implies the cost of the join operation itself. The cost of inserting and expiring tuples is ignored because it is not influenced by join ordering. The cost of
join operation over data stream is rather different from traditional database. For relational data join, only the I/O cost is contained (CPU cost is much less than I/O cost). Instead of storing in the disk, the data streams directly enter the input buffer then are insert to the sliding window for processing. The whole join procedure is performed in the memory. Thus only the CPU cost is calculated over data stream.

The data structure of sliding window can be hash table or queue, which is mapped to hash join or nested loop join respectively. Hence based on the unit-time cost model [9], the cost formulas of nested loop join and hash join about sliding window are shown as:

$$\mathcal{W}_j = \text{Cost}(\theta_j)$$

$$= |W_j| \cdot |W_i| \cdot |W_j| \cdot |C_n|$$

$$= \lambda_j T_j \lambda_i T_i M \cdot M_j M_i C_n$$  \hspace{1cm} (6)

$$\mathcal{W}_j = \text{Cost}(\theta_j)$$

$$= |W_j| \cdot |W_i| \cdot |W_j| \cdot |C_n|$$

$$= \lambda_j T_j \lambda_i T_i M \cdot M_j M_i C_n$$  \hspace{1cm} (6)

$$\mathcal{W}_j = \text{Cost}(\theta_j)$$

$$= |W_j| \cdot |W_i| \cdot |W_j| \cdot |C_n|$$

$$= \lambda_j T_j \lambda_i T_i M \cdot M_j M_i C_n$$  \hspace{1cm} (6)

$$\mathcal{W}_j = \text{Cost}(\theta_j)$$

$$= |W_j| \cdot |W_i| \cdot |W_j| \cdot |C_n|$$

$$= \lambda_j T_j \lambda_i T_i M \cdot M_j M_i C_n$$  \hspace{1cm} (6)

3.3 Graph-based join algorithm

Using the graph join model, we can deal with sliding window multi-joins. Figure 1 shows the graph-based sliding window join algorithm with lazy re-evaluation. As pointed out earlier, there are three phases in our algorithm: founding join model, selecting join query plan and executing query plan.

In the first phase, when setting up the SWWDJG before probing in step 03, the new arrival tuples in stream i are seemed as a new \(W_i\) to replace the original one, so that the output result can be updated continuously with optimal join order. This is similar to the pipelining algorithm [4] that the new arrival

\begin{verbatim}
Input: A continuous sliding window multi-join query
Output: The result tuples
Begin: Every time the query is re-executed
01: Insert new tuples into sliding windows;
02: For i=1...n;
03: Set up the SWWDJG;
04: JoinOrder (SWWDJG); Return \(W_i\Theta_iW_i...W_{n-1}W_n\)
05: \(\forall k \leq n\) and \(\forall n \geq k\)
   If \(t_k.\text{attr} \leq t_1.\text{attr} \ldots \ldots \) loop from \(W_n\) to \(W_k\)
06: \(W_k \in W_n\) and \(W_n = \text{NOW} \ldots \text{and} \) \(t_1 - t_k \leq \text{NOW} \) and \(t_k - t_n \leq \text{NOW} \) and \(t_k - t_i \leq \text{NOW} \) and \(t_k - t_n \leq \text{NOW} \) and \(t_k - t_i \leq \text{NOW} \)
   If \(t_k.\text{attr} \leq t_1.\text{attr} \ldots \ldots \ldots \ldots \ldots \ldots \ldots \)
07: Return \(\theta_1 \theta_2 \theta_3 \ldots \theta_n\)
08: Endif
09: Endif
10: End

(Figure 1) The Graph-based Join Algorithm
\end{verbatim}
tuples are inserted to its corresponding sliding window, and then probe other windows.

In the second phase, we use the improved MVP algorithm [6] to find an optimal join query plan in step 04, as the cost of nested loop join about sliding window is different. In original MVP algorithm an inflowing edge $v_i \rightarrow v_j$ whose weight $w_{v_i \rightarrow v_j}$ is less than 1 can reduce the cost of vertex $v_j$. That means if the intermediate result replaces the original relation the tuple number of join result will be reduced. But when processing sliding window, from (6) and (7) we can see the two weight vectors influence same to join cost. Thus the inflowing edge whose weight $w_{v_i \rightarrow v_j}, w_{v_j \rightarrow v_i}$ is less than 1 can reduce the cost of sliding window join.

In the last phase, the query plan is executed through a nested loop join procedure where the timestamp of tuples have to satisfy certain condition to make sure the result is valid.

3.4 Example

In a four time-based sliding windows join query, $\theta_1$ is on $w_1$ and $w_2$, $\theta_2$ on $w_2$ and $w_3$, $\theta_3$ on $w_3$ and $w_4$, $\theta_4$ on $w_4$ and $w_1$. Assume that the tuple size of each sliding window is 100, and their parameters are shown in Table 2.

Using the graph-based algorithm we first set up the corresponding SWWJDG in Figure 2. Also the vertex weight (cost of each join operator) is calculated (here we assume $c_n = 1$).

<table>
<thead>
<tr>
<th>$\theta$</th>
<th>$JSF_i$</th>
<th>$JCF_i$</th>
<th>$\lambda_i$</th>
<th>$T_i$</th>
<th>$\lambda_{i+1}$</th>
<th>$T_{i+1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\theta_1$</td>
<td>0.002</td>
<td>0.5</td>
<td>10</td>
<td>100</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>$\theta_2$</td>
<td>0.001</td>
<td>0.1</td>
<td>2</td>
<td>100</td>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>$\theta_3$</td>
<td>0.05</td>
<td>0.2</td>
<td>5</td>
<td>100</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>$\theta_4$</td>
<td>0.005</td>
<td>0.5</td>
<td>1</td>
<td>100</td>
<td>10</td>
<td>100</td>
</tr>
</tbody>
</table>

$\overline{W}_1 = \text{cost}(\theta_1) = \lambda_1 T_1 \lambda_2 T_2 M_1 M_2 C_n = 2 \times 10^9$
$\overline{W}_2 = \text{cost}(\theta_2) = \lambda_2 T_2 \lambda_3 T_3 M_2 M_3 C_n = 1 \times 10^9$
$\overline{W}_3 = \text{cost}(\theta_3) = \lambda_3 T_3 \lambda_4 T_4 M_3 M_4 C_n = 0.5 \times 10^9$
$\overline{W}_4 = \text{cost}(\theta_4) = \lambda_4 T_4 \lambda_1 T_1 M_4 M_1 C_n = 1 \times 10^9$

\[ \text{Figure 2} \] The SWWJDG of the Example Join

In this graph, a near optimal join sequence is selected by using the improved MVP algorithm which gives the result: $v_2 \rightarrow v_1 \rightarrow v_4 \rightarrow v_3$. The comparison among different join orders is shown below. Obviously the join order found by the MVP algorithm is optimal.
(Table 3) Possible Join Sequence and Their Cost

<table>
<thead>
<tr>
<th>Effective Spanning Tree</th>
<th>Total execution cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4$</td>
<td>$6.44 \times 10^9$</td>
</tr>
<tr>
<td>$v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_1$</td>
<td>$5.32 \times 10^9$</td>
</tr>
<tr>
<td>$v_3 \rightarrow v_4 \rightarrow v_1 \rightarrow v_2$</td>
<td>$30.125 \times 10^9$</td>
</tr>
<tr>
<td>$v_4 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3$</td>
<td>$3.2 \times 10^9$</td>
</tr>
<tr>
<td>$v_4 \rightarrow v_3 \rightarrow v_2 \rightarrow v_1$</td>
<td>$17 \times 10^9$</td>
</tr>
<tr>
<td>$v_1 \rightarrow v_4 \rightarrow v_3 \rightarrow v_2$</td>
<td>$7.4 \times 10^9$</td>
</tr>
<tr>
<td>$v_3 \rightarrow v_2 \rightarrow v_1 \rightarrow v_4$</td>
<td>$3.77 \times 10^9$</td>
</tr>
<tr>
<td>$v_2 \rightarrow v_1 \rightarrow v_4 \rightarrow v_3$</td>
<td>$1.72 \times 10^9$</td>
</tr>
</tbody>
</table>

4. Performance evaluation

The evaluation environment and comparison among our join algorithm (GRAPH), Lazy Multi-Way NLJ (LAZY) [10] algorithm and General Lazy Multi-Way NLJ (GEN) [10] algorithm are presented in this chapter. In LAZY algorithm, the join sequence is ordering the new arrived tuples first. The join order in GEN is specified to select the joining with minimal $JSF$ first.

4.1 Evaluation environment

We implement our algorithm by using VC programming, which runs on Windows PC with a 1.8GHz Pentium(R) 4 processor and 1GB physical memory.

The data streams are produced by the following way. In each iteration of a loop $\frac{i}{n}$ stream $i$ is selected with probability $\sum_{j=1}^{n} \lambda_j$, while a tuple is generated for the selected stream. Each tuple includes three attributes: the system-assigned timestamp $ts$, the integer data $d$ and the filled part. The integer data is selected from corresponding data set $\{d, \ldots, a\}$. The $JSF$ of two windows is calculated as $\frac{1}{\max(a_i, a_j)}$. Towards $JCF$, the output tuple size is adjusted by increasing or reducing the filled part. For each case, certain tuples are generated and the processing time measured. Each case is run many times and the average result is reported.

4.2 Evaluation results

In the first experiment, we specify the window size, value set and $JCF$. The relative rates of two streams are set certainly, while new streams are specified. Figure 3 shows that LAZY is better than GRAPH at the beginning, but GRAPH outperforms as the number of window increases. The reason is when the number of windows is low ordering newly arrived tuple first is always optimal so that GRAPH has to spend extra time finding query plan. As the number of windows increases the time saving from executing query plan is much more than the time to find the join order. GRAPH is much better than GEN. That is because $JSF$ in this experiment is fixed so that GEN selects the join operator naively. Besides, GRAPH not only considers $JSF$ but also $JCF$.

In the second case, the four sliding window has been joined with parameters in table 2 which is evaluated when varying re-evaluation intervals. From Figure 4, it can be observed that the LAZY and GENERAL algorithms are more expensive than GRAPH with optimal query plan.
5. Conclusion

In this paper, we proposed a new graph model, named SWWDJG, which can properly represent all the execution orders about sliding window multi-join in a standard way. Many significant parameters have been used in the graph to calculate the join cost and to gain optimal query plan. Then we present a new sliding window join algorithm by using the model to specify the sliding window join order. As the query plan selects the optimal join sequence, the performance can be improved in a greater manner.

In future, we can find more efficient algorithms to provide query plan based on the graph model. As graph-based join algorithm assumes memory that is big enough to store all data in sliding window, we can also research load shedding within limited memory.

6. References

7. A. N. Wilschut, P. M. G Apers, "Dataflow

강 령 (Zhang Liang)
2004년 Department of Electronic Engineering, Nanjing University of Posts and Telecommunications, China (공학사)
2005년 현재: Department of Computer Science, Chongqing University of Posts and Telecommunications, China (석사과정)
관심분야: stream data query processing, Spatial Database, Spatial Information & Data Management

유병섭 (ByeongSeob You)
2002년 인하대학교 전자·전기·컴퓨터공학부-컴퓨터공학(학사)
2004년 인하대학교 대학원 전자계산공학과(공학석사)
2008년 인하대학교 대학원 정보공학과(박사예정)
관심분야: 공간 데이터베이스, 공간 데이터 웹상, 센서 네트워크

거준위 (Jinwei Ge)
1991년 geophysical technology, Chengdu University of Technology, China (학사)
1991년 현재 a professor in the Faculty of Software at Chongqing University of Posts and Telecommunications, China
관심분야: software requirements engineering, software architectures, GIS

김경배 (GyoungBae Kim)
1992년 인하대학교 전자계산공학과(공학사)
1994년 인하대학교 대학원 전자계산공학과(공학석사)
2000년 인하대학교 대학원 전자계산공학과(박사학위)
2000년-2004년 한국전자통신연구원 선임연구원
2004년-현재 서원대학교 컴퓨터교육과 조교수
관심분야: 이동통신시간데이터베이스, 스토리지시스템, GIS, VOD

이순조 (SoonJo Lee)
1985년 인하대학교 전자계산공학과(학사)
1987년 인하대학교 대학원 전자계산공학과(학사)
1995년 인하대학교 대학원 전자계산공학과(박사학위)
1997년-현재 서원대학교 컴퓨터정보통신공학부 부교수
관심분야: 데이터베이스 시스템, 정보보안, GIS

배혜영 (HaeYoung Bae)
1974년 인하대학교 음용물리학과(공학사)
1978년 연세대학교 전자계산공학과(학사)
1990년 서울대학교 전자계산공학과(박사학위)
1992~1994년 인하대학교 전자계산소 소장
1992년-현재 인하대학교 컴퓨터공학부 교수
1999년-2007년 지능GIS연구센터 설탕장
2000년-현재 한국중앙무선단 대학원 명예교수
2004년-2006년 인하대학교 정보통신대학원 원장
2006년-현재 인하대학교 대학원 원장
관심분야: 분산 데이터베이스, 공간 데이터베이스, 저리정보 시스템, 멀티미디어 데이터베이스