Design and Implementation of Bit-Vector filtering for executing multi-join queries

By

Cheng Bin

Department of Computer Science
School of Computing
National University of Singapore

2008/09
Design and Implementation of Bit-Vector filtering for executing of multi-join queries

By
Cheng Bin

Department of Computer Science
School of Computing
National University of Singapore

2008/09

Project No: U024050
Advisor: Dr. CHAN Chee Yong
Deliverables:
  Report: 1 Volume
  Source Code: 1 CD
Abstract

In this project, we investigate the bit vector techniques on the hashjoin and scan operators of the multi-join queries. In the previous work, bit vectors have been applied to drop non-matching tuples from joining relations before the execution of a join, thus reducing the join cost. Because the bit vectors built in different execution stages can have different costs, we apply the bit vectors constructed on the upper join node to the lower scan node to further reduce the traffic of the tuples. We use TPC-H benchmark to generate 1GB database and queries, which are running on an open source database management system—PostgreSQL. We explain how bit vector works, implement the algorithm and evaluate its effectiveness and tradeoffs.

Keywords:
bit vector, hybrid hash-join, sequential scan, PostgreSQL

Implementation Software and Hardware:
PostgreSQL 8.3.5
Acknowledgement

I would like to thank my supervisor Dr. CHAN Chee Yong who guides me to experience research work and provide a great help for me.

I would also like to thank my friends Chen Yu, Li hao and Li hongyang who discuss and share ideas with me. Without them, I would not have be able to complete this project.
# List of Figures

2.1 An example of query plan ................................. 4
3.1 sample query plan .................................. 9
4.1 Query 18 ........................................ 12
4.2 Query plan tree for query 18 ............................... 13
4.3 Execution time with all the bit vectors turned on ............. 13
4.4 Tuple Traffic with all the bit vectors turned on ............... 14
4.5 Execution time with some bit vector turned off ............... 15
4.6 Tuple traffic with some bit vector turned off ............... 15
4.7 Query 2 .......................................... 17
4.8 Query plan tree for query 2 ............................... 18
4.9 Execution time of query 2 ................................. 18
4.10 Tuple Traffic with all the bit vectors turned on ............... 19
4.11 Overhead and effectiveness of bit vectors .................... 20
List of Tables

4.1 Query 18: hashjoin clause of hashjoin node ........................................... 11
4.2 Query 2: hashjoin clause of hashjoin node ............................................. 16
4.3 Tuple traffic of all the scan nodes ....................................................... 19
Table of Contents

Title .............................................. i
Abstract ........................................... ii
Acknowledgement ................................... iii
List of Figures .................................... iv
List of Tables ..................................... v

1 Introduction .................................... 1
  1.1 Background .................................... 1
  1.2 The Problem ................................... 2
  1.3 Our Solution ................................... 2
  1.4 Report Organization ............................ 2

2 Study of the PostgreSQL database system ......... 3
  2.1 System Architecture ............................ 3
  2.2 Executor Engine and Query Plan ............... 3

3 Implementation .................................. 6
  3.1 Data structure ................................ 6
  3.2 Implementation Details ........................ 7
    3.2.1 ExecutorStart ............................ 7
    3.2.2 ExecutorRun .............................. 8
    3.2.3 ExecutorEnd .............................. 9

4 Evaluation ..................................... 10
  4.1 Experimental Setup ............................ 10
    4.1.1 Software Setup ........................... 10
    4.1.2 Hardware Setup ........................... 11
  4.2 Query 18 ..................................... 11
    4.2.1 Query Description ........................ 11
    4.2.2 Traffic and Execution Time ............... 14
  4.3 Query 2 ....................................... 16
    4.3.1 Query Description ........................ 16
    4.3.2 Traffic and Execution Time ............... 18
5 Conclusion

5.1 Contributions ................................................................. 21
5.2 Future Work ................................................................. 22

References ................................................................. 23
Chapter 1

Introduction

The execution time of queries to the relational database system is greatly influenced by the I/O. In parallel systems, bit vector filters have been used very effectively for what we call here probabilistic semi-joins. (Graefe, 1993). Although it is an old idea to apply bit vector filtering to speed up query execution, there is very little work on evaluating its effectiveness and tradeoffs.

1.1 Background

The problem we study in this report is an interesting one. Since bit vector is an efficient tool to reduce the execution time of multi-join queries, it has been used in commercial DBMSs like DB2 (Graefe, Bunker, & Cooper, 1998), Oracle (Gongloor & Patkar, 1997) and SQLServer (Graefe et al., 1998). However, PostgreSQL has not implemented the bit vector yet, so we are going to apply the bit vector on PostgreSQL to study its performance.

This problem is first proposed in 1997 (Chen, Hsiao, & Yu, 1997). Chen gives the first algorithm to the problem and applied the bit vector to the sort-merge join of a bushy execution tree of multi-join queries. More recently, a slightly different formulation of the problem is studied independently (Josep, Victor, & Pey, 2003). Although Josep’s paper is an improvement over chan’s paper, the query he used is a simple one. In our project, we test our code on slightly more complex queries to analysis the effectiveness and tradeoffs.
1.2 The Problem

Joins are among the most expensive operators in a relational database system, and their implementation has a big impact on performance. In this project, we are to minimize the execution time of multi-join queries. The algorithm we are studying is the hybrid hash join algorithm. Suppose we are going to join two relations, R as the outer relation and S as the inner relation. First when we scan the inner relation S, we hash S into numbers of batches and build hash table for each batches. We do the same for the outer relation R, instead of build hash table, we probe tuples in R into the hash table to see if there are any matches, if not, we drop the tuple immediately.

We can see that there are lots of I/O incurs during the reading of batch files produced by the scan of relations. We try to reduce the I/O by introducing bit vector to drop tuples in the early phase of execution.

1.3 Our Solution

In this project, we are building bit vector on the build phase of hybrid hash-join phase and apply them to the sequential scan of the relations.

The bit vector comes in when we first scan the inner relation S, we build a global bit vector for it, on the hashjoin node. Then when we first scan the outer relation R, we probe each tuple into the bit vector to see if there are any matches. After applying the bit vector, the tuples delivered to the hash join node can be greatly reduced which saves I/O time.

1.4 Report Organization

Our report consists of 4 parts. Chapter 1 gives an introduction of the problem. Chapter 2 gives us an overview of the PostgreSQL architecture. Chapter 3 presents our detailed solution to the hybrid hash-join algorithm and sequential scan operation. Chapter 4 tries to evaluate the performance of bit vector technique turned on/off with different sizes.
PostgreSQL is an open source relational database system. PostgreSQL’s source code is available to all, and gives us the opportunity to use, modify and distribute PostgreSQL in any form we like. As such, PostgreSQL is not only a powerful database system capable of running the enterprise, it is a development platform upon which we can modify and add in new features for our own research purpose.

2.1 System Architecture

There are five main components in the PostgreSQL architecture. The parser parses the query string entered by the end user; the rewriter applies the rewrite rules; the optimizer chooses an efficient query plan among many possible plans; the executor executes the best query plan; the utility processor processes DDL like CREATE TABLE.

In this project, we will have a close look at the executor engine of PostgreSQL. The input to the executor engine is a query plan, and consists of a tree of nodes, with each node denoting a query operator. The bit vector technique applies to the hash join operator and sequential scan operator to reduce the execution time of the query plan.

2.2 Executor Engine and Query Plan

The optimizer choose the best query plan and pass it to the executor. A sample query plan tree is stated in Figure 2.1.
Two main operators for the executor engine is the join operator and scan operator, and each operator is shown as a node is the query plan tree. The function of the various nodes is shown below:

1. Sequential scan node is to scan the relation sequentially, without using index.

2. Hash node fetches a tuple from the child scan node and hashes the tuple into the hashtable. If the relation is too big, multiple hashtables are produced, and hashtables are loaded into memory one at a time.

3. Hash join node invokes the hash node to build hashtables and fetches tuple from the child scan node to see whether there is a hit in the hashtable, if so, it compares the actual value and produces a tuple if there is a match.

We can see that the execution starts from the root of the query plan tree, which invokes the execution of its right and/or left child. The execution also ends at the root node when there are no results to be produced by its children.

It’s common for the user to issue nested query to the system. PostgreSQL will attach sub plans to the main query plan tree that is necessary to handle the nested query. It is not required to generate sub plans for the nested query if the rewriter can applying the rewrite rules to simplify the query plan.

The sub plan may be stores in the hashjoin clause of the hashjoin node, in the form of "subplan = B.m". For example, hash join node S has a hashjoin clause with two entries: "A.m = B.n" and "subplan = C.x". PostgreSQL has two different functions to evaluate these two types of hashjoin entries.

When hashjoin entry "A.m = B.n" is evaluated, the hashjoin node will extract attribute m from the
pull out and probe into the hashtable. When hashjoin entry "subplan = C.x" is evaluated, it will invoke the subplan to produce the tuple and extract the relevant attribute to probe into the hashtable.

Hash join node retrieve tuple from its child scan node one at a time. The returned attribute information of every node is stored on the target list of each node, which contains information about from which child node does the attribute come from and the attribute number of the attribute. These information is very useful when we build and apply bit vectors
Chapter 3

Implementation

This project is mainly focus on the study of the PostgreSQL source code, understand the database system architecture, extract the relevant information, implement our algorithm on the original engine and study the performance. While there are only a few source files need to be modified, most of the time spending on this project is on studying the source code and implementing the bit vector technique on it. In this chapter, we provide the data structure for the bit vector and the implementation details for building and applying bit vectors.

3.1 Data structure

Bit vector is first proposed in 1970 (Bloom, 1970) and it’s called Bloom Filter. We follow the convention and create the data structure BloomData and denote type Bloom to be the pointer to BloomData.

typedef struct BloomData{
    int asize;
    int *a;
    MemoryContext cxt;
} BloomData;

typedef struct BloomData *Bloom;

asize is the size of the bit vector. We have two versions of bit vectors with size 1K and 8K, which is set accordingly when we run the experiments later to see the effectiveness of bit vectors with different
sizes.

PostgreSQL provides a convenient way to manage memory allocation. Unlike the conventional malloc routine, PostgreSQL allocate memory within memory context. When bit vector is created, PostgreSQL allocates memory from the current memory context, while destroying a memory context releases all the memory that was allocated in it. It efficiently prevents memory leaks.

3.2 Implementation Details

Given the input query plan tree from the optimizer, our implementation modifies the code of the executor engine to apply bit vector technique to execute the query. The files of the executor engine is located under the executor folder. The file execMain.c contains the top level executor interface routines: executorStart(), executorRun() and executorEnd().

3.2.1 ExecutorStart

ExecutorStart routine is called at the beginning of execution of any query plan. It initializes the query plan to produce the query plan state tree, which is the query plan tree in execution. If the query plan tree consists of sub plans, the sub plans are initialized first before the initialization of the main query tree. This is necessary because the information of the sub plan is required for the main plan.

The bit vectors are created at this stage. We traverse from the root of the query plan tree to find all the hash join nodes. Each node in PostgreSQL has a nodeTag field indicating the type of the node. We use nodeTag to find the hash join node, if any hash join node is found, we extract the relevant information from the hash join clauses of the node. The hash join clauses have a list of hash join entries, with each entry in the form of "m = n". m is the attribute name of outer relation while n is the attribute name of inner relation. We create a bit vector for each hash join clause entries, so each hash join node has a list of bit vectors with it.

In order to build bit vector on attribute n and apply it on m, we need to locate the attribute m in the base relation. The varattno member of an attribute structure indicates from which target list entry does the attribute comes from. Once we locate the correct entry in its child’s target list, we apply the same procedure recursively to locate the attribute in the base relation. We also need to figure out from which
child does the attribute comes from: left child or right child. Then we create a link between the attribute in the seqscan node and the corresponding bit vector, so later when we scan the base relation, we can apply the bit vector on the correct attribute.

The bit vectors can be turned on/off as indicated in a disk file. If the bit vector is turned off, no bit vector is created for this particular attribute and there is no link between the seqscan node the hashjoin node.

If any hashjoin node has sub plans with it, we always turn off bit vectors to be created at this hashjoin node because the bit vector cannot link between main query and sub query.

3.2.2 ExecutorRun

ExecutorRun is the main routine of the executor module. After the various nodes have been initialized appropriately and bit vectors have been created and linked, we start to execute the query. The root node is invoked first to retrieve tuples in the specified direction. In order for the root node to produce tuples, it retrieve tuples from its children and perform some computations on them. The child node been called by the parent node retrieve tuples from its children and the same procedure is repeated recursively until we have processed all the tuples.

Take Figure 3.1 for example. The upper hash-join node is called to produce the final results. It starts with the building phase of hybrid hash-join algorithm. Because we have already created a bit vector on attribute $a$ of relation $R3$, when we build hashtables on $R3$, we insert the hashvalue into the bit vector at the same time. Once the build phase is completed, the hashjoin node invokes the left child to retrieve the tuples. When the execution reaches the scan node of relation $R2$, it will look up the bit vector on attribute $a$ to filter out tuples when scanning relation $R2$. After applying the bit vector, the output cardinality of the scan node is reduced, which in turn, reduces the amount of data to be processed by the join node. Also, the output cardinality of the join node is reduced and so is the amount of date to be processed by the upper hashjoin node. If relation $R1$ also has attribute $a$, it can also look up the bit vector to further reduce the traffic.
Figure 3.1: sample query plan

3.2.3 ExecutorEnd

ExecutorEnd is called at the end of the execution of any query plan. The memory allocated to various node and the bit vector are released. If it has sub plan in the query plan, the nodes in the sub plan tree are released first before the free of the main plan nodes.
Chapter 4

Evaluation

In this section, we present the result using different parameter settings. The evaluation results measure the traffic volume and execution time. Before running the query, we explain the software and hardware setting for the experiment.

4.1 Experimental Setup

4.1.1 Software Setup

We have generated a 1GB database using TPC-H benchmark. We implement our bit vector technique on PostgreSQL. We have two versions of PostgreSQL; the original engine without bit vector and our own implementation with bit vector. We also run the experiments with different sizes of bit vector and turn on/off the bit vector to get further insight of the effectiveness and tradeoffs.

We implement the bit vector technique on the hash join and sequential scan operators, so we turn off all other join and scan operators to force the engine to generate query plan with no other join and scan operators that will confuse our study. The memory space allocated to each hash join node is called sortmem in PostgreSQL. We follow the default sortmem size in the configuration file.

A profiler tool, gprof, is used to study the execution time spend in each function, which shows the possible speedup and overhead of our implementation.
<table>
<thead>
<tr>
<th>Hashjoin node</th>
<th>Hashjoin Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td>Lineitem.l.orderkey = orders.o.orderkey</td>
</tr>
<tr>
<td>H2</td>
<td>Orders.o.custkey = customer.c.custkey</td>
</tr>
<tr>
<td>H3</td>
<td>Orders.o.orderkey = Lineitem.l.orderkey</td>
</tr>
</tbody>
</table>

Table 4.1: Query 18: hashjoin clause of hashjoin node

### 4.1.2 Hardware Setup

We run all the experiments on a 1.66G Hz Duo-core machine with 1GB of main memory, 80G hare disk. The operating system is Debian 2.6.

### 4.2 Query 18

#### 4.2.1 Query Description

Each query in the PostgreSQL is internally represented as a query plan tree. Figure 4.1 is query 18 from the TPC-H benchmark and it is translated into query plan tree stated in Figure 4.2, indicating how the PostgreSQL engine executes the query.

Basically, each relation in the `from clause` has a seqscan node associated with it, and each join predicate in the `where clause` has an entry in the hashjoin clauses of the corresponding hashjoin node. The hashjoin clauses of the hashjoin nodes H1, H2, H3 are shown on Table 4.1.

The number of bit vectors required in our implementation is the same as the number of all hashjoin clause entries. In this case, we build three bit vectors on attribute orders.o.orderkey, customer.c.custkey and Lineitem.l.orderkey. Those bit vectors are applied to Lineitem.l.orderkey, Orders.o.custkey and Orders.o.orderkey, respectively. It is indicated as the dashed line in the query plan tree on Figure 4.2. For example, there are two bit vectors that can be applied to the `orders` relation, from hashjoin node H2 and H3. SeqScan node $S1$ can also make use of the bit vector constructed on the hashjoin node H1.
select  
c_name,  
c_custkey,  
o_orderkey,  
o_orderdate,  
o_totalprice,  
sum(l_quantity)  
from  
customer,  
orders,  
lineitem  
where  
o_orderkey in (  
    select  
l_orderkey  
    from  
    lineitem  
    group by  
l_orderkey having  
    sum(l_quantity) > 300  
)  
and c_custkey = o_custkey  
and o_orderkey = l_orderkey  
group by  
c_name,  
c_custkey,  
o_orderkey,  
o_orderdate,  
o_totalprice  
order by  
o_totalprice desc,  
o_orderdate;  

Figure 4.1: Query 18
Figure 4.2: Query plan tree for query 18

Figure 4.3: Execution time with all the bit vectors turned on
4.2.2 Traffic and Execution Time

Figure 4.3 and Figure 4.4 show the execution time and tuple traffic with three different settings. We compare the statistics from executing of the original code, the bit vector with size 1K and bit vector with size 8K. The tuple traffic for each seqscan node shows the number of tuples delivered to the upper node, from which we can have an insight on the effectiveness of the bit vector technique.

From the figures, we can see that we have improved the execution time with bit vector of size 1k and 8k. The original PostgreSQL implementation without any modification shows the running time of 235268.897ms. We achieve 27% improvement by using bit vectors of size 1K and we get a better improvement by setting the size of bit vector to be 8K, which is 35% compared with the original code.

The major improvements in the traffic are obtained by the scan of relation Lineitem (S1) and relation orders (S2). This is because the bit vector created in the corresponding hashjoin node has significant efficiencies in query execution. When the bit vectors are been applied on the seqscan node, the Lineitem table and orders table is filtered out significantly, reducing the traffic delivered to the upper nodes. There are no traffic reduction for node S3 and S4 because no bit vectors can be applied to these two nodes.

We can also turn on/off the bit vectors to see which bit vector has most great impact on the query execution. Figure 4.5 and Figure 4.6 show the execution time and tuple traffic with the three bit vectors turned off one at each time.

Figure 4.5 and Figure 4.6, we can see that if we turn off the bit vector at the HashJoin node H1,
Figure 4.5: Execution time with some bit vector turned off

Figure 4.6: Tuple traffic with some bit vector turned off
<table>
<thead>
<tr>
<th>Hashjoin node</th>
<th>Hashjoin Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td>part.p_partkey = partsupp.ps_partkey</td>
</tr>
<tr>
<td>H1</td>
<td>subplan = partsupp.ps_supplycost</td>
</tr>
<tr>
<td>H2</td>
<td>partsupp.ps_suppkey = supplier.s_suppkey</td>
</tr>
<tr>
<td>H3</td>
<td>supplier.s_nationkey = nation.n_nationkey</td>
</tr>
<tr>
<td>H4</td>
<td>nation.n_regionkey = region.r_regionkey</td>
</tr>
<tr>
<td>H5</td>
<td>partsupp.ps_suppkey = supplier.s_suppkey</td>
</tr>
<tr>
<td>H6</td>
<td>supplier.s_nationkey = nation.n_nationkey</td>
</tr>
<tr>
<td>H7</td>
<td>nation.n_regionkey = region.r_regionkey</td>
</tr>
</tbody>
</table>

Table 4.2: Query 2: hashjoin clause of hashjoin node

the improvement reduced to 5%, compared with the original implementation. The reason for this is that HashJoin node $H1$ is the most crucial part for the I/O optimization. At around 44% of the input tuples is from the SeqScan node $S1$, which is assigned bit vector by HashJoin node $H1$. With the bit vectors turned on at size 1K(8K), the result shows that at around 34%(43%) of total tuples are reduced orders.o_orderkey for the upper node execution, which greatly reduce the execution time. The figure also tells that the bit vector created at hashjoin node $H2$ does not help to filter out any tuple at the seqscan node $S2$. So we should always turn off this bit vector to reduce the overhead.

### 4.3 Query 2

#### 4.3.1 Query Description

We have studied the execution time and tuple traffic of query 2 with various bit vectors. In this section, we are going to study a more complex query involving a sub plan. The query is in Figure 4.7 and the corresponding query plan tree is stated in Figure 4.8.

There are seven hash join nodes in the main plan and sub plan. In all, we create 6 bit vectors and we don’t create bit vectors in hash join node $H1$ because it has a sub plan in its hash join clause. The hash join clauses as shown on Table 4.2.
select
    s_acctbal, s_name, n_name, p_partkey,
    p_mfgr, s_address, s_phone, s_comment
from
    part, supplier, partsupp,
    nation, region
where
    p_partkey = ps_partkey
and s_suppkey = ps_suppkey
and p_size = 15
and p_type like '%BRASS'
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'EUROPE'
and ps_supplycost = (    
    select
        min(ps_supplycost)
    from
        partsupp, supplier, nation, region
    where
        p_partkey = ps_partkey
    and s_suppkey = ps_suppkey
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = 'EUROPE'
)
order by
    s_acctbal desc, n_name, s_name, p_partkey;

Figure 4.7: Query 2
4.3.2 Traffic and Execution Time

Figure 4.9, Figure 4.10 and Table 4.3 compares the execution time and tuple traffic between the original code, our code with all the bit vectors turned on with size $1k$ and all the bit vectors turned on with size $8k$.

Figure 4.9 shows that the execution time of bit vectors with size $1k$ has increased to 1037406.999 ms.
Figure 4.10: Tuple Traffic with all the bit vectors turned on

Table 4.3: Tuple traffic of all the scan nodes

<table>
<thead>
<tr>
<th>Scan node</th>
<th>Original</th>
<th>1K</th>
<th>8K</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>20000</td>
<td>20000</td>
<td>20000</td>
</tr>
<tr>
<td>S2</td>
<td>800000</td>
<td>701760</td>
<td>205040</td>
</tr>
<tr>
<td>S3</td>
<td>10000</td>
<td>1988</td>
<td>1988</td>
</tr>
<tr>
<td>S4</td>
<td>25</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>S5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>S6</td>
<td>965601206</td>
<td>847025655</td>
<td>247485238</td>
</tr>
<tr>
<td>S7</td>
<td>10000</td>
<td>1988</td>
<td>1988</td>
</tr>
<tr>
<td>S8</td>
<td>25</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>S9</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
compared with the original code, which executes in 902401.033 ms. Figure 4.10 and Table 4.3 shows that the 99% of the total tuple traffic is from the scan of relation \textit{Partsupp} at seqscan node S6. Although the bit vector has reduced the tuple traffic by 12.4%, the execution time is increased by 15%.

![Figure 4.11: Overhead and effectiveness of bit vectors](image)

The bit vectors with size 8K has filter out the tuple traffic by 74.4% and reduced the execution time by 19.3%. Figure 4.11 shows the overhead and the improvement of the bit vectors from various functions.

1. Function \textit{SeqNext} is the workhorse for sequential scan operation. It shows that the execution time of \textit{SeqNext} increased by using bit vectors, and this is because that in order to build the bit vectors, we have to extract the relevant attribute from the scan tuple and this takes much time in our implementation.

2. Function \textit{ExecBloomGetHashvalue} is used to check whether this is a hit for the given tuple in the bit vector. The overhead is incurred when we apply the bit vector to drop tuples.

3. Function \textit{ExecQual} is called by the hash join node to see whether the tuple satisfies the condition of the hash join clause. Since large percentage of tuples have been dropped by the bit vectors, the number of time that \textit{ExecQual} is called is reduced, which reduces the overall execution time.
Chapter 5

Conclusion

5.1 Contributions

In this paper, we have implemented the bit vector technique on the hybrid hash-join algorithm and the sequential scan operator to study its effectiveness and tradeoffs. The bit vector is created in the upper hash join node of the query plan and is applied on the sequential scan operator to reduce the tuple to be delivered to the upper node.

We outline the following conclusions for our implementation:

1. Different size bit vectors can be created to utilize the amount of memory being allocated to the bit vectors.

2. Bit vectors can be turned on and off depending on its usefulness for the query execution.

3. Our implementation reduces the total amount of the tuple traffic significantly, which has a direct influence on the workload of the nodes.

4. Since the amount of the tuple traffic is greatly reduced, the amount of I/O is also reduced, which is a bottleneck of the query execution.

5. The total execution time of the query is reduced, which is the main concern for the commercial DBMSs.
5.2 Future Work

Bit vector is a useful tool to improve the query execution. The idea doesn’t only work for the hybrid
hash-join algorithm and sequential scan operation, and it can also be applied to other join and scan
operations, such as sort-merge join and index scan operations.

Bit vector can also be introduced by the optimizer engine to determine the best query plan. It can be
treated as an operator to work with all other join and scan operators in the query plan tree.

Since some commercial DBMSs like Oracle has already implemented the bit vector technique on it,
we hope PostgreSQL can also implement and integrate bit vector to improve the system performance.


