Instruction Retentiveness to increase Query Performance through Cache Management – with Column Store

Tejaswini Apte  
Devi Ahilya VishwaVidyalay, Indore, India  
Email: apte.tejaswini@gmail.com

Maya Ingle and A. K. Goyal  
Devi Ahilya VishwaVidyalay, Indore, India  
Email: {maya_ingle@rediffmail.com, goyalkcg@yahoo.com}

Abstract—Cache management has strong impact on database query performance. Literature survey has shown, tuning the database cache may improve performance significantly. Materialization of intermediates in database cache plays vital role in improving the response time. We propose an enhancement to a column store database structure by building additional intelligence in the form of an adaptive and efficient query cache, called KPCS (Keep Pool Column Store). The proposed algorithm is evaluated against the TPC-H schema to demonstrate effectiveness in terms of significant performance gains.

Index Terms—TPC-H, column store

I. INTRODUCTION

The need of good response time of DSS queries has given birth to materialization and reuse of intermediates before eviction from cache. The materialization of intermediates can be avoided in tuple-at-a-time pipelined execution model thereby in minimizing the opportunities for shared and reused computation. To overcome with this limitation, materialized view or query-result-set-caches may be used by commercial database systems. Operator-at-a-time execution paradigm promotes the exploitation of intermediates to speed up queries [1], [2], [3]. Materialized views represent common sub-queries, in which materialization improves subsequent time of processing. Recycle of intermediates is useful in applications for parameterized queries, with minimum eviction rate.

As, it has been observed that frequent eviction from cache affects the hit ratio badly [4], [5]. However, it is possible to improve cache hit rate by introducing Keep-Pool in column store database architecture. Keeping the intermediate in the cache improves response time and throughput problems and as a whole it refines caching methodology used in column store database system. Experiments have been conducted against the operator-at-

II. RELATED WORK

Keeping the intermediates enhances the generic idea of storing and reusing expensive computations. Keeping the results is also the fundamental idea of materialized views and query caching [8], [9]. To improve the system performance materialized views have defined by the DBA [10]. The dynamic materialized views refreshes as soon as data changes. Traditionally, view or intermediate matching is integrated with the query optimization. By the eviction strategy the system adapts the pool content to maximal benefit of the current workload [11].

III. KEEP-POOL ARCHITECTURE (KPA)

The architecture of the Keep-Pool is targeted to SQL queries over read-only database. Each query plan is produced in isolation, i.e. without knowledge of workload itself using query optimizer techniques. Thesults of operation from previous queries are not taken into account at optimization time. Instead, matching of instruction, followed by decisions to reuse it, is performed at run time.

A. Design Instructions for Keep-Pool

The first design step is to identify instructions to the Keep-Pool, performed during query optimization by the
Keep-Pool optimizer. It inspects the plan and marks instructions to be controlled by Keep-Pool. The cheap instructions are not kept in the Keep-Pool. The overhead of administration outweighs the expected gain. Keep-Pool threads access the columns of SQL query and propagates the property through query plan. Typically, the threads involve selections, joins and other primary relational operations. The Keep-Pool adapts continuously to the workload without DBA intervention and incurs minimum start-up and maintenance cost. The design contains three dimensions as Query matching, execution cost and pool administration.

B. Keep-Pool Policies

To improve the cache performance, Keep-Pool acts as a buffer for storing instruction, parameters and results. The instructions are stored with three parameters namely: Reuse statistics, CPU time to compute, and Result size. The admission policy determines the candidate instruction for Keep-Pool, thus preventing the eviction rate. The instructions are evicted from Keep-Pool to manage the memory resource, to prevent overflow, resulted in the cache miss for costlier instruction. Entries are evicted from the pool when the base relation for that instruction changes.

C. Proposed Algorithm-KPCS

In this section, we present the flow of KPCS with informal and formal description.

D. Informal Description

KPCS is broadly designed with three functions: Query (TPC-H-Q) is a built in function that accepts TPC-H query and returns a valid instruction as output, as a valid instruction is required for query execution. Match (I) is also a built in function, designed to check the presence of an instruction in Keep-Pool. The function keepEntry(I) is developed to check the instruction availability in Keep-Pool. keepPool(I) The candidate instruction is admitted to the Keep-Pool by said procedure. The built-in function freeSpace(KP) maintains the heap of available space. The focus of keepExit(I) is to track the execution frequency and evict instructions from Keep-Pool.

E. Formal Description – KPCS

/* It will keep the frequently used instruction in the memory in order to reuse it for improving the performance */

/* Main program begins */

void main()
{
    Instruction I;
    I=Query (TPC-H);
    Call keepPool(I);
}
/* End of main */

Procedure keepPool/Instruction I
/* This procedure keeps candidate instruction to Keep-Pool */

{ /* Variable Declaration */
    Heap kp;
    int frequency;

    /* Allocating 10% of 1GB RAM */
    kp=malloc((1024)*3*10/100);
    if marked(I) then
        if not keepEntry(I) then
            execute I;
            kp=kp-sizeof(I);
            call keepExit(I);
        else
            execute I;
        end if;
    else
        execute I;
    end if;
}
/* End of Procedure keepPool */

Boolean function keepEntry/Instruction I /* This function checks for instruction availability in the keep pool, if available execute the same from Keep-Pool */

{ /* Variable Declaration */
    Boolean M;
    /* Function call to check instruction in Keep-Pool*/
    M=Match(I) ;
    if M then
        execute I;
        return true;
    else
        return False;
    end function;
}
/* End of function keepEntry */

Procedure keepExit/Instruction I /* This procedure checks the execution frequency then evicts instruction from Keep-Pool */

{ /*Variable Declaration */
    Heap kp;
    int threshold=40;

    /*Checking the available space*/
    if admission(I) then
        if sizeof(I)<freeSpace(kp) then
            if marked(I) then
                if execute(marked(I))>=threshold then
                    /* Clean cache by flushing least recently used instruction*/
                    cleanInstructionCache (sizeof(I));
                    add I to kp;
                end if;
            end if;
        end if;
    end if;
}
/* End of Procedure keepExit */
IV. MAINTENANCE OF KEEP-POOL

Normally, it has been found out that the database server is overloaded while checking the instructions and for testing the candidature for eviction. The process is supported by the frequency of execution of an instruction. The factors affecting the capture rate of an instruction for eviction are: Instruction execution frequency and Percentage of performance gain.

The performance gain is measured by:
- Resource cost the system has spent to compute results.
- The reuse weight (number of reuses).

When resource limit is reached, candidates are to be evicted from Keep-Pool. Resource limit can be measured in terms of memory or number of entries in the Keep-Pool.

V. TPC-H EVALUATION

To understand the Keep-Pool mechanism better and its effect on performance, we have conducted the experiments with TPC-H decision support benchmark [12]. KPCS has been executed on 10GB database and the measurements have been taken against the hot cache. For this benchmark we executed 10 instances of each query generated with the TPC-H query generator. To illustrate the Keep-Pool mechanics better, the admission policy is keep-all and there are no resource limitations, thus no cache policy interferes with the results. Queries have been analyzed with respect to commonalities, benefited from reusing intermediates. As the number of commonality increases, the hit ratio improves, with degradation in resource utilization. Result may vary with respect to commonality in queries. We have observed the time improvements and memory consumption after each query as shown in Table I, which is driven by commonality in queries. The targeted queries for these experiments are simple, join and aggregated queries. The Fig. 1 shows the execution time of individual queries, which is directly proportional to hit ratio. To estimate the query commonalities we assume that the same query is executed with different parameters, where the parameter generation follows the TPC-H specification. Query Q9 contains substantial intra-query commonality where a large part of the sub-query is shared with the main query. The common part includes a selection, a 2-way join, and an arithmetic computation over projected attributes. Since the query overlap is negligible, the time improvement and the hit ratio are stable for all instances. The cache policy has a different influence over different types of queries. The small candidate instructions are always put in the pool and reused for current query execution approximately always, hence prevent eviction. The eviction policy, evicts instructions based on frequency of usage in Keep-Pool. Obstacle for performance improvements are huge complex queries. The huge complex queries which cannot fit in the Keep-Pool result in eviction of small instructions, which directly lead to higher number of Keep-Pool misses and performance degradation. A manual inspection of the execution traces of queries demonstrating high time savings, queries Q4, Q7, and Q21, confirmed that the reused instructions have lowered the execution cost drastically. For other queries, such as Q17, time savings are minimal despite the high percentage of instructions reused.

<table>
<thead>
<tr>
<th>TPC-H Queries</th>
<th>Initial execution time (in ms)</th>
<th>KPCS execution time (in ms)</th>
<th>Initial memory consumption (in kb)</th>
<th>KPCS memory consumption (in kb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>306.015</td>
<td>79.59</td>
<td>59,176</td>
<td>57,278</td>
</tr>
<tr>
<td>3</td>
<td>900</td>
<td>459.335</td>
<td>61,172</td>
<td>60,164</td>
</tr>
<tr>
<td>4</td>
<td>3100</td>
<td>566.817</td>
<td>61,428</td>
<td>60,252</td>
</tr>
<tr>
<td>5</td>
<td>1700</td>
<td>889.573</td>
<td>65,964</td>
<td>63,460</td>
</tr>
<tr>
<td>6</td>
<td>614.468</td>
<td>332.213</td>
<td>66,072</td>
<td>64,692</td>
</tr>
<tr>
<td>7</td>
<td>11800</td>
<td>975.637</td>
<td>65,912</td>
<td>64,784</td>
</tr>
<tr>
<td>8</td>
<td>2400</td>
<td>233.075</td>
<td>70,428</td>
<td>69,040</td>
</tr>
<tr>
<td>9</td>
<td>3500</td>
<td>782.187</td>
<td>80,608</td>
<td>78,160</td>
</tr>
<tr>
<td>10</td>
<td>4000</td>
<td>2000</td>
<td>1,09,736</td>
<td>1,06,980</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>760.684</td>
<td>1,12,928</td>
<td>1,08,632</td>
</tr>
<tr>
<td>16</td>
<td>857.605</td>
<td>589.215</td>
<td>1,12,288</td>
<td>1,09,840</td>
</tr>
<tr>
<td>17</td>
<td>404.548</td>
<td>320.771</td>
<td>1,12,784</td>
<td>1,11,120</td>
</tr>
<tr>
<td>19</td>
<td>3500</td>
<td>2000</td>
<td>1,18,064</td>
<td>1,13,836</td>
</tr>
<tr>
<td>21</td>
<td>6700</td>
<td>900</td>
<td>1,21,761</td>
<td>1,19,200</td>
</tr>
</tbody>
</table>

VI. RESULT ANALYSIS AND CONCLUSION

In this paper we have described a database architecture augmented with Keep-Pool instructions. The architecture is implemented as an extension to the MonetDB system. The Keep-Pool policies respect the inter-operator dependencies, which leads to effective reuse of large threads in template based query sessions. The policies cover an LRU scheme and resource cost. The challenges are, to decide the factors for the retention period and
eviction strategy, for optimal utilization of resources. To evaluate the cache policies, we have used 14 TPC-H queries as shown in Table I. The unlimited space strategy has been used to measure the total resources needed. We have considered two major areas for resource consumption namely; memory consumption and instruction matching time. The experiments have been performed on machine running RHEL 5 on a 2.4 GHz Intel processor and 1GB of RAM. As observed, the average improved time with Keep-Pool becomes approximately four times better than the previous architecture. The architecture is affected by the amount of I/O and processing bandwidth available in the system, with selectivity and number of columns accessed as parameters as shown in Fig. 1. The extensive experimentation based on a full-edged implementation shows that the MonetDB software architecture is well suited to be extended with a targeted optimization goal. The results obtained indicate several areas for further exploration. Within the context of MonetDB, it seems worth exploring sub-summation relationships through join paths and the opportunities offered by recognition of query classes. Another direction of work is refining and developing admission and cache policies that respect the semantic dependencies of instructions, as well as automatic accommodation of policies most appropriate for the current workload.

![Figure 1. KPCS versus Initial time consumption](image)

REFERENCES


Maya Ingle did her Ph.D in Computer Science from DAU, Indore (M.P.) . M.Tech (CS) from IIT, Kharagpur, INDIA, Post Graduate Diploma in Automatic Computation, University of Roorkee, INDIA, M.Sc. (Statistics) from DAU, Indore (M.P.) . INDIA. She is presently working as PROFESSOR, School of Computer Science and Information Technology, DAU, Indore (M.P.) . INDIA. She has over 100 research papers published in various International / National Journals and Conferences. Her areas of interest include Software Engineering, Statistical, Natural Language Processing. Usability Engineering, Agile computing, Natural Language Processing, Object Oriented Software Engineering, interest include Software Engineering, Statistical Natural Language Processing, Usability Engineering, Agile computing, Natural Language Processing, Object Oriented Software Engineering. Dr. Maya Ingle has a lifetime membership of ISTE, CSI, IETE. She has received best teaching Award by All India Association of Information Technology in Nov 2007.

©2013 Engineering and Technology Publishing
Tejaswini Apte received her M.C.A (CSE) from Banasthali Vidyapith, Rajasthan. She is pursuing research in the area of Machine Learning from DAU, Indore, (M.P.), INDIA. Presently she is working as an ASSISTANT PROFESSOR in Symbiosis Institute of Computer Studies and Research at Pune. She has 4 papers published in International/National Journals and Conferences. Her areas of interest include Databases, Data Warehouse and Query Optimization. Mrs. Tejaswini Apte has a professional membership of ACM.

Arvind Goyal did his Ph.D in Physics from Bhopal University, Bhopal (M.P.), M.C.M from DAU, Indore, M.Sc.(Maths) from U.P. He is presently working as SENIOR PROGRAMMER, School of Computer Science and Information Technology, DAU, Indore (M.P.). He has over 10 research papers published in various International/National Journals and Conferences. His areas of interest include databases and data structure. Dr. Arvind Goyal has lifetime membership of All India Association of Education Technology.