

Accelerating Analytics with Dynamic In-Memory Expressions

Aurosish Mishra, Shasank Chavan, Allison Holloway, Tirthankar Lahiri, Zhen Hua Liu, Sunil Chakkappen, Dennis Lui, Vinita Subramanian, Ramesh Kumar, Maria Colgan, Jesse Kamp, Niloy Mukherjee, Vineet Marwah

Oracle America
400 Oracle Parkway
Redwood Shores CA 94065
aurosish.mishra@oracle.com

ABSTRACT

Oracle Database In-Memory (DBIM) accelerates analytic workload performance by orders of magnitude through an in-memory columnar format utilizing techniques such as SIMD vector processing, in-memory storage indexes, and optimized predicate evaluation and aggregation. With Oracle Database 12.2, Database In-Memory is further enhanced to accelerate analytic processing through a novel lightweight mechanism known as *Dynamic In-Memory Expressions* (DIMEs). The DIME mechanism automatically detects frequently occurring expressions in a query workload, and then creates highly optimized, transactionally consistent, in-memory columnar representations of these expression results. At runtime, queries can directly access these DIMEs, thus avoiding costly expression evaluations. Furthermore, all the optimizations introduced in DBIM can apply directly to DIMEs. Since DIMEs are purely in-memory structures, no changes are required to the underlying tables. We show that DIMEs can reduce query elapsed times by several orders of magnitude without the need for costly pre-computed structures such as computed columns or materialized views or cubes.

1. INTRODUCTION

Oracle Database In-Memory (DBIM) provides extensive optimizations for accelerating most aspects of analytic workloads, including scans, joins, predicate evaluation and aggregation [3]. Each of these query components involves expressions, the evaluation of which is often the dominant cost of query execution [5]. For instance, consider the following query:

```
SELECT item_name,  
       price * (1 - discount)  
FROM   sales  
WHERE  category = 'household';
```

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.

Proceedings of the VLDB Endowment, Vol. 9, No. 13
Copyright 2016 VLDB Endowment 2150-8097/16/09.

If the SALES table is in the in-memory columnar format, it can be scanned and filtered for 'household' items at the rate of billions of rows per second. The evaluation of the SELECT expression `price * (1 - discount)`, on the other hand, involves costly numerical computations which can slow down overall query execution.

Common approaches for reducing expression evaluation costs include adding pre-computed columns to base tables, or creating materialized views or pre-defined cubes. All of these are typically difficult to define for ad-hoc workloads, and are expensive to maintain when the underlying tables change frequently. If a query repeats the same expression multiple times, common sub-expression elimination (CSE) [8] can be used to evaluate each expression only once. The results, however, are not cached from one query to another, so subsequent queries cannot take advantage of the evaluation.

This paper introduces *Dynamic In-Memory Expressions* (DIMEs), a novel lightweight mechanism that identifies expensive expressions and caches them "on the fly" in-memory, allowing queries to access them at runtime, thus avoiding redundant expression evaluations. Our proposed mechanism begins by automatically tracking all expressions evaluated across a query workload in a repository known as the *Expression Statistics Store* (ESS). Frequently executed, costly expressions are selected from the ESS, and then computed and cached in the In-Memory (IM) Column Store. Subsequent query execution involving the same captured expressions are optimized in the scan engine by directly accessing the cached results from memory, side-stepping the expression evaluation engine entirely for those expressions.

It should be noted that DBIM features a dual-format in-memory architecture, where the persistent data format remains row-oriented for efficient OLTP performance, and a pure in-memory columnar format is used to accelerate analytic workloads. Thus, it is possible to add additional expression evaluation result columns to the in-memory columnar format without having to make any changes to the underlying physical tables – the expressions materialized by the DIME infrastructure only exist in the IM column store.

The rest of this paper is organized as follows. Section 2 provides a brief overview of DBIM and the organization of the IM column store in terms of In-Memory Compression Units (IMCUs). Section 3 introduces the DIME concept and how candidate

expressions can be identified using the ESS. Section 4 describes how DIMEs are populated in terms of units known as In-Memory Expression Units (IMEUs) and how they are maintained as the underlying table data changes. Section 5 describes how DIMEs are used to accelerate scans, predicate evaluation and aggregations involving expressions. Section 6 provides some experimental results and Section 7 concludes.

2. OVERVIEW OF ORACLE DATABASE IN-MEMORY

Row stores are ideal for OLTP workloads, where each transaction typically accesses a small number of rows and many columns in each row (e.g insertion of a new order). On the other hand, column stores [1,2] are better suited for analytics workloads in which queries access many values in a small number of columns (e.g. find the number of sales in each state).

Since neither format is optimal for all workloads, Oracle Database In-Memory supports a dual-format in-memory representation [3] in which the row format continues to be supported via the buffer cache, while a new pure in-memory columnar format is added for the subset of tables on which fast analytics is required (see Figure 1). The in-memory columnar format is a pure in-memory format, therefore no logging or check-pointing is required for its maintenance as the underlying row data changes. DML changes run directly against the row format, and the column format is transactionally maintained. Highly selective OLTP-style queries (e.g. lookup by primary key) are directed by the Optimizer [7] to use the row format, while analytic queries are directed by the Optimizer to use the column format – regardless of which format is used, the same results are returned by the query.

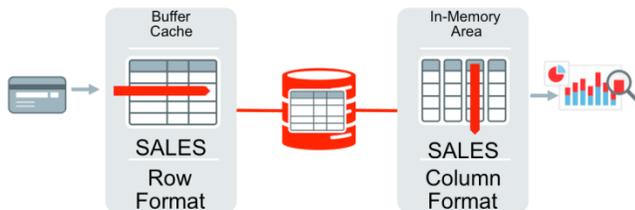


Figure 1. Dual-Format In-Memory Database

The IM column store can be used for all or a subset of the tables in a database. When a table is brought into the column store, it is done by a process known as *Populate*, which creates the column formatted version of the table from its underlying row format. The Populate process creates the in-memory column formatted table in terms of units known as *In-Memory Compression Units* (IMCUs). An IMCU spans a large range of rows, between 0.5-1 million, and within each IMCU, columns are organized into *column Compression Units* (CUs) – which are large compressed vectors of column values. A variety of compression schemes are available, depending on the user-chosen level of compression (e.g. it is possible to choose a compression scheme optimized for maximum query performance, or one that is optimized for maximum space savings).

Scans against the IM column store are optimized using *SIMD vector processing* instructions, which can process multiple operands in a single CPU instruction (see Figure 2) [3,4]. Further, each IMCU maintains per column summary information such as minimum and maximum values. This collective summary

information serves as an *in-memory storage index* for the table. It allows IMCUs to be skipped completely while processing a table scan, when it is known from the storage index that none of the rows in the IMCU will qualify based on the scan filter predicates. Thus, by reducing the amount of data accessed per scan, faster query response times are achieved.

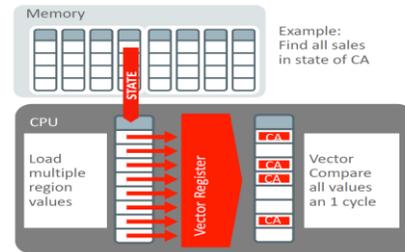


Figure 2. SIMD vector-processing in the IM column store

Each IMCU is associated with a *Snapshot Metadata Unit* (SMU) that tracks changes made by DMLs since the time of creation of the IMCU. Oracle Database employs a snapshot-based isolation model known as *Consistent Read* (CR) [6]: Each operation is associated with a snapshot *System Change Number* (SCN) representing when the operation began, and is only allowed to see either its own changes or changes that were committed at earlier SCNs. The SMU provides CR semantics: scans consult both the IMCU and the SMU in order to generate consistent results. When the number of changes to an IMCU exceeds a certain threshold (the threshold is determined by a combination of heuristics) a *Repopulate* task is issued on the IMCU to create a pristine version once again.

3. IDENTIFYING CANDIDATE EXPRESSIONS

Expressions are essential components of analytic SQL queries. They can involve simple mathematical or logical operators such as “+” and “*” as well as built-in SQL functions such as *substr()*, *regexp()*, *trunc()* and user-defined PL/SQL functions. Expressions can occur in various parts of a SQL statement. For example, expressions may exist within the SELECT list, WHERE clause predicates, an aggregation function, and within the GROUP BY and HAVING clause.

Table 1. SQL queries with expressions

ID	QUERY
Q1	<code>SELECT (sal + bonus) FROM emp</code>
Q2	<code>SELECT SUM((sal + bonus)*(1-taxrate)) FROM emp WHERE UPPER(job) = 'MANAGER'</code>
Q3	<code>SELECT MAX(sal) FROM emp GROUP BY EXTRACT(year FROM hiredate)</code>

In the examples shown in Table 1, Q1 contains the expression `(sal + bonus)` in the SELECT list, Q2 contains the expression `(sal + bonus)*(1-taxrate)` inside an aggregation, and another expression `UPPER(job)` in the WHERE clause predicate, and finally, Q3 contains the expression `EXTRACT(year FROM hiredate)` in the GROUP BY clause.

An expression can be completely subsumed by another expression. Such an expression is referred to as a *sub-expression*. An expression that is a composite of multiple sub-expressions is termed a *top-level expression*. For example, Q2 has the top-level expression $(sal + bonus) * (1 - taxrate)$, which contains two sub-expressions: $(sal + bonus)$ and $(1 - taxrate)$.

The examples above focus on expressions that are explicitly invoked in SQL queries. However, the SQL engine often generates *implicit expressions* and internal computations during query compilation. As an example, consider the where clause: $c1=c2$. The query optimizer may choose to rewrite this predicate as $c1-c2=0$, thereby generating the implicit expression $c1-c2$. Data conversions, hash computations, column concatenations, etc. are all examples of implicit computations that can be generated internally by the SQL engine to help speed up complex analytic queries and join operations.

The evaluation of implicit and explicit expressions can consume a significant amount of CPU time within an analytic workload. Furthermore, the same expressions may recur across several different queries in a given workload. By automatically identifying such expressions and materializing their results in-memory, the database can greatly improve query performance while reducing the amount of CPU resources consumed. The benefits of in-memory materialization are two-fold – a) avoid repeated expression evaluations and b) apply in-memory query optimizations such as SIMD vector processing, in-memory storage index pruning, etc. on the materialized expression results.

A DIME can be broadly classified as any expression that is automatically captured from the workload, and for which the results have been pre-computed and materialized within the IM column store. Strictly speaking, a typical DIME involves one or more columns of a table, possibly with some constants and has a 1-to-1 mapping with the rows in the table. In order to identify DIMEs, we build and maintain a repository capturing useful statistics about expressions that are evaluated in various queries of an analytic workload, called the Expressions Statistics Store (ESS).

3.1 Expression Statistics Store

The Expression Statistics Store is a database-level repository maintained by the optimizer that tracks statistics of various expressions received and generated by the SQL engine on a per-table basis. The ESS identifies expressions to track for a query at compilation time. There are many different stages during query compilation that can transform, eliminate or add new expressions. The expression tracking mechanism is deferred till after the compile time representation of the expression is finalized. This ensures that expressions are in their final form and will not undergo any further transformations.

To uniquely identify expressions across different queries, the ESS generates an *expression ID* for each distinct expression of a table. The expression ID is a unique encoding obtained from the canonical representation of the expression and the table object number. For any expression, the canonical form is generated after normalizing the expression by transforming it in different ways, including commutative, associative and distributive transformations. This ensures that two expressions on the same table, such as $(a+b)$ and $(b+a)$, that differ in their textual representation but have the same canonical form, are tracked as the same expression in the ESS.

Factoring the table object number into the encoding function ensures that expressions are tracked on a per table basis in the ESS. For instance, if two tables T1 and T2 have an expression $upper(c)$ on a varchar2 column c , they will be treated as different expressions in the ESS. For PL/SQL procedures, the expression ID is generated by encoding the PL/SQL package ID and the package entry number for the procedure.

Each expression tracked by the ESS is associated with two distinct types of attributes: Static attributes and Dynamic attributes. Static attributes include information that is fixed for a particular expression and does not change across different query executions. These include the SQL text representation of the expression, list of columns referenced in the expression, optimizer fixed cost that estimates the per-evaluation processing cost of the expression, etc. Dynamic attributes track information that changes from one query to another. They include expression evaluation counts, timestamps of expression evaluation, optimizer dynamic costs based on runtime feedback, etc.

During query execution, the most accurate method to track evaluation count of an expression is to keep counters in the evaluation procedures. However, this is fairly involved and may cause performance regressions in critical query paths. Hence, different heuristic-based approaches are employed to estimate the evaluation count for expressions.

One simple heuristic utilizes *row source statistics*. A row source in Oracle corresponds to a node in a query execution plan. It is an iterative control structure that accepts a set of rows from child nodes, processes them in an iterated manner, and produces an output row-set for its parent node. The SQL engine has several row sources such as the table scan row source, various join method row sources, partition iterator row sources, etc. For each such row source, the row source statistics contain information about the number of rows flowing in and out of that row source. These numbers are used to estimate the most likely evaluation count of an expression within that row source.

Each row source can provide run-time feedback to the ESS to more accurately estimate dynamic attributes such as the expression evaluation count and execution cost. This can be done by annotating the row source statistics with information about the actual number of rows processed per expression, or number of expression evaluations pruned by a certain expression occurring in a predicate. For example, the table scan row source may receive two expressions (e.g. $round(price)$ and $upper(item_name)$) as part of two different predicates (e.g. $round(price)=10$ and $upper(item_name) = 'COFFEE'$) in the WHERE clause. As part of the first predicate evaluation, a large fraction of the rows may be filtered out, causing the expression in the second predicate to be evaluated for only a small number of rows. This fine-grained information is only available inside the row source, but is essential for the accuracy of tracked expression statistics.

Both static and dynamic attributes for expressions are stored in the shared memory within the System Global Area (SGA), which is a per-instance read/write memory area that is shared by all processes belonging to that Oracle instance [14]. This information is also persisted periodically to separate dictionary tables on disk to ensure that expression statistics tracked by the ESS are durable across database restarts.

The ESS maintains run-time statistics for different time-horizons, in separate *snapshots*, in order to provide greater flexibility in statistics monitoring. For example, two intuitive snapshots

supported by the ESS are: *cumulative* and *current*. The cumulative snapshot contains expression statistics since the first time an expression was captured by the ESS (e.g. cumulative evaluation count), while the current snapshot captures execution statistics within the last N hours (e.g. last 24 hours). The current snapshot statistics are merged into the cumulative snapshot statistics once the expression creation timestamp crosses the N hour mark.

3.2 DIMEs and ESS

3.2.1 Candidate Expression Ranking

The ESS tracks various statistics and metadata for all candidate expressions in a database workload. However, the goal of the DIME infrastructure is to capture *hot* expressions that account for a significant fraction of the total evaluation cost. The hotness of an expression essentially represents the cumulative cost incurred by the SQL engine in evaluating that expression repeatedly across different queries. Each expression is given a weighted *hotness score* that depends on a number of factors such as evaluation count, dynamic execution cost, row source in which the expression occurs, distribution of expression evaluation across different snapshots, etc. In addition, statistics captured across different snapshots may be weighed differently. For example, statistics captured in the current snapshot can be given a higher weight than statistics in the cumulative snapshot to ensure that recently seen expressions are considered more favorably. Using this hotness score, expressions are ranked for a particular table or across the entire database. A simple formula to compute the hotness score of an expression is shown in Figure 3.

$$\forall e, H_e = \sum_{snap=1}^n W_{snap} \cdot C_{snap}$$

$$C_{snap} = f(c_e, cnt_e, sid, \dots)$$

$$\sum_{snap=1}^n W_{snap} = 1$$

$$W_{snap} \in [0,1]$$

where:

- H_e , hotness score of expression e
- W_{snap} , weight given to a snapshot
- C_{snap} , cost of evaluating e in that snapshot
- f , evaluation cost function for e that depends on:
 - c_e , average execution cost per evaluation
 - cnt_e , evaluation count in a snapshot
 - sid , id of snapshot to consider

Figure 3. Hotness score for expressions

The snapshots and benefit function can be fine-tuned as required to provide better ranking of expressions.

The DBIM infrastructure has an in-memory coordinator process (IMCO), which periodically queries the ESS, ranks expressions based on their hotness score, and obtains the set of “top n ” expressions at the database level. These expressions are populated into the IM column store as DIMEs.

The number of expressions chosen depends on several factors. Obviously, materializing all expressions would provide the greatest performance benefits across the widest range of queries. However, we have to weigh the query performance benefits of DIMEs against their increased memory footprint in the IM column store. Factors such as compression format chosen, data-

types of base columns in the expression, and the amount of in-memory space available for expressions, are all taken into account while deciding on the number of DIMEs to capture from the ESS.

3.2.2 Virtual Columns

For any hot expression captured from the ESS, and eventually stored in the in-memory area, we need a unique way to identify it across various layers of the SQL engine. A simple way to achieve this is to leverage the *virtual columns* infrastructure of Oracle Database.

Virtual columns (VCs), introduced in Oracle 11g, are columns that represent expressions on one or more table columns. Unlike a base column (physical column), a VC is represented only as table metadata – it does not have any physical allocation on disk. When queried, its value is computed by evaluating the expression at runtime. Any reference to a VC is automatically replaced with its expression in the logical expression tree and tagged with a special flag by the SQL engine. Similarly, any occurrence of an expression, which is represented by a VC, is also tagged with the same flag, enabling the SQL engine to identify VCs during query execution.

The DIME infrastructure adds hot expressions captured from the ESS as *hidden VCs* to the respective tables. Hidden VCs differ from user-defined VCs in that they are not visible to the user, and are not returned as part of a ‘SELECT *’ or a DESCRIBE query on the table. DIME hidden VCs are also assigned a separate system-generated namespace to distinguish them from user-defined VCs. This enables the DIME infrastructure to add and remove hidden VCs from a table automatically without user intervention. The addition and removal of a hidden VC are lightweight operations that do not affect running applications.

The list of hot expressions returned by the ESS can change as the workload generates newer expressions. Hence, cold expressions must be removed to prevent unnecessary consumption of in-memory space. In each lookup of the ESS, we mark the DIME hidden VCs that have become cold to be ‘*no inmemory*’ using the selective columns feature (see Section 3.3). Alternatively, we can mark the cold hidden VCs as ‘*unused*’, since unused columns are never chosen for in-memory materialization.

3.3 User-Defined Virtual Columns

The DIME infrastructure relies on the ESS to automatically capture hot expressions from an analytic workload. Once identified, these expressions are added to the table as hidden VCs. Naturally, another source of candidate expressions includes user-defined VCs. The techniques used to accelerate queries using DIMEs can be directly applied to user-defined VCs as well. Thus, for completeness, we provide a manual counterpart of the DIME feature referred to as *In-Memory Virtual Columns*.

This manual component of the DIME feature provides users with a greater degree of control over which virtual columns to populate into the IM column store. For example, users can specify a column compression clause on each VC denoting whether or not they want the VC to be stored in-memory, and at what in-memory compression level. Similar to base columns, users can choose from multiple compression levels for VCs: FOR DML, FOR QUERY, and FOR CAPACITY. [3,4]

For the remaining sections, we focus on DIMEs only, noting that the same framework can be used for populating, maintaining and querying user-defined VCs as well.

4. CREATION AND MAINTENANCE OF DIMES

DIMES are materialized in special in-memory units called *In-Memory Expression Units* (IMEUs). The memory for storing IMEUs comes from the same In-Memory Area in the SGA reserved for the IM column store. IMEUs utilize the same in-memory columnar format as the base table. Recall that each table selected for in-memory storage is populated into the IM column store in contiguously allocated units called IMCUs [3,4]. An IMEU is implemented as a logical extension of an IMCU. The IMCU, which an IMEU logically extends, is referred to as the *parent IMCU*. Physically, an IMEU is stored as a top-level continuation piece of the parent IMCU, with a pointer from the IMCU to the IMEU. Storing DIMES in separate IMEUs, rather than storing them within the IMCU has several advantages, as described over the next few sections.

Each IMCU stores column data for a target number of table rows, typically half a million. The IMEU stores DIME results for each of those rows stored in the parent IMCU (see Figure 4). Within the IMEU, each DIME is stored contiguously as an *Expression Unit* (EU). EUs utilize in-memory formats, similar to those used for column CUs in the parent IMCU.

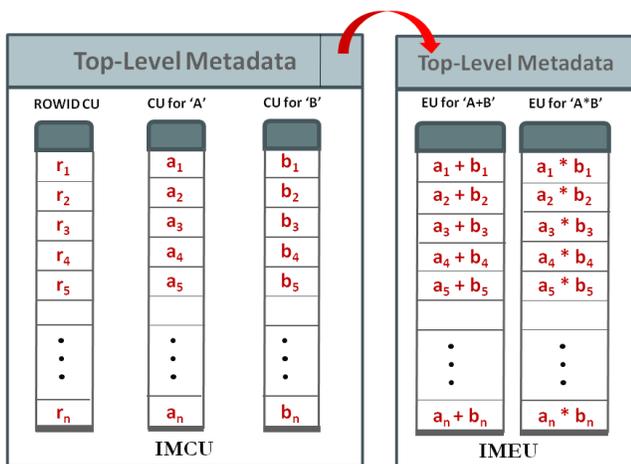


Figure 4. Columns A and B in an IMCU with n-rows and expressions A+B and A*B in the corresponding IMEU

An IMEU inherits all in-memory attributes from the parent IMCU and the on-disk table/segment that was used to populate the IMCU. For example, the IMEU is duplicated or distributed in a RAC configuration [3] in exactly the same manner as the parent IMCU. Thus, the distribution manager can provide the same high-availability, fault-tolerance and scalability guarantees for IMEUs, as for IMCUs. Similarly, IMEU population is performed in the same priority order as the parent IMCU, as specified by the PRIORITY sub-clause on the base table. The data in the IMEU is typically also compressed using the same compression schemes used to populate the parent IMCU. In certain cases, EUs may be compressed at higher compression levels (such as FOR CAPACITY) to ensure maximum space utilization.

4.1 Population of IMEUs

IMEUs utilize the same background population mechanism that is used to build the IMCUs. The IMCO coordinates population tasks using a configurable pool of background server processes. Each

population task contains metadata about which set of on-disk rows to populate in a particular IMCU. Since an IMEU spans the same set of on-disk rows as the IMCU, the population task context is simply augmented with the list of DIMES to populate. Thus, an IMCU and its IMEU are both populated by the same background process as part of the same population task (see Figure 5). This guarantees that all the concurrency control primitives that synchronize IMCU population with DDL operations such as ALTER/DROP TABLE, DROP TABLESPACE, etc. will now synchronize IMEU population as well.

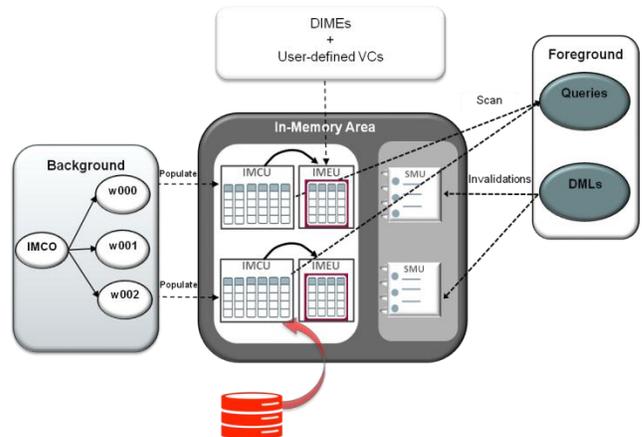


Figure 5. Top-level IMEU population (with IMCU and SMU)

Each background slave first completes the IMCU population by column formatting rows obtained from a subset of on-disk blocks for the table, and applying appropriate compression schemes. The SMU is also built in this process to track transactional changes for the rows in this IMCU. Once the IMCU is populated, it is deemed online, and queries can access column data from the IMCU. Similarly, transactions can also proceed and DMLs will be recorded in the SMU to track validity of the IMCU rows. Only after the IMCU is online, does IMEU population begin. This ensures that applications have no downtime in accessing IMCUs and SMUs, even when there are IMEUs to be populated.

Unlike IMCU source data that is readily available in the row-format, the IMEU data, i.e. DIME results, is not available in the row-store. Hence, the row data is used to evaluate the expressions to generate DIME results. Subsequently, intelligent data transformations and compression algorithms are applied on this data to create DIME EUs for the IMEU.

As mentioned in Section 2, each IMCU is marked with the SCN of the time of its creation. The IMCU contains all committed changes up-to that SCN for the rows it spans. Any changes beyond that SCN are tracked in the SMU. To ensure transactional consistency with the IMCU, the IMEU is built as of the IMCU creation SCN using Oracle point-in-time queries, known as *flashback queries*.

Introduced in Oracle 9i, flashback queries employ CR techniques to view past states of database objects without using point-in-time media recovery [9]. To fetch DIME results, an internal AS OF SCN flashback query is issued, with the query SCN being same as the IMCU creation SCN. Since the query SCN and the IMCU creation SCN match, and the IMEU spans the same set of rows as the IMCU, the flashback query performs expression evaluation

directly on the column CUs in the recently built parent IMCU. Thus, using flashback queries guarantees that contents of an IMCU are consistent with source data within the parent IMCU.

Any row of a table has base column data in the IMCU and corresponding DIME data in the IMEU. Hence, the same SMU that tracks validity of rows in the IMCU can be leveraged for tracking transactional changes in the IMEU. Thus, any query accessing data from the IMEU is guaranteed to always obtain consistent expression results.

4.2 Re-population of IMEUs

Once IMEUs are populated, queries containing expressions that have been materialized as DIMEs, can directly access the expression results from the EUs. However, for rows invalidated by DMLs, the expression results cannot be directly read from the IMEU, and must be computed during runtime. Naturally, as DMLs accumulate, performance of DIME scans deteriorates just as it does for scans on the IMCU. Hence, we employ a background *repopulate* mechanism to periodically ‘refresh’ the IMCU-IMEU pair and rebuild it at a new SCN.

As described in [3], IMCUs are repopulated using two techniques: threshold-driven repopulation and trickle repopulation. A variety of policies are employed to control threshold-driven repopulation. Repopulation thresholds take into account the number of invalid rows/blocks in an IMCU, number of scans on an IMCU, etc. Once any IMCU exceeds a certain threshold, it is queued for repopulation. Trickle repopulation, unlike threshold-driven repopulation, runs constantly and unobtrusively in the background, consuming a small fraction of the available repopulate processes. The goal of trickle repopulation is to ensure that eventually any given IMCU is clean even if it has not exceeded the repopulation thresholds.

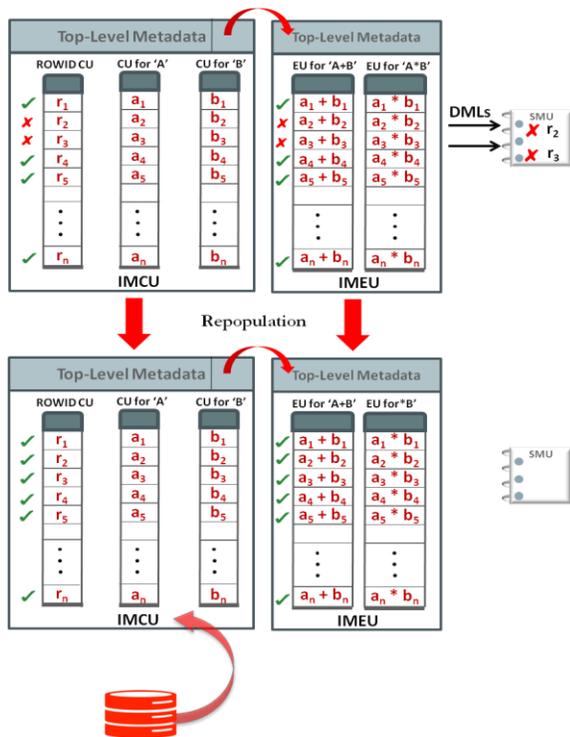


Figure 6. Repopulation of IMCU-IMEU

From their creation, IMEUs are tightly coupled with their parent IMCUs. Thus, IMEUs are repopulated whenever the parent IMCU is repopulated as a result of DMLs on the base table (see Figure 6). IMEUs must also be repopulated whenever we want to add a new hot DIME, or evict a cold DIME. This mechanism is an IMEU-only repopulation; the parent IMCUs need not be repopulated when the set of hot expressions being tracked by the ESS changes. This is one of the advantages of storing expressions in separate IMEUs, and not mixing them inside IMCUs.

For any repopulation operation, the old IMCU and IMEU are kept online until the new IMCU and IMEU have been created. This ensures that applications do not suffer a significant drop in performance due to IMCU-IMEU unavailability.

During each lookup of the ESS, if the set of hot expressions has changed significantly, proactive repopulation tasks are submitted to remove cold DIMEs and populate new hot DIMEs in-memory. This guarantees that only the analytic working set of expressions are materialized as DIMEs in-memory at any point in time, thereby achieving maximum performance benefits with optimum memory utilization.

5. LEVERAGING DIMEs FOR QUERY ACCELERATION

Once candidate expressions are identified and hidden VCs are created to represent them, DIMEs are populated into IMEUs and become fully accessible for query optimization. The next step involves rewriting the query plan generated by the SQL compiler into a runtime execution plan that can leverage DIMEs during expression evaluation. When this plan is processed by the scan engine, VCs look practically identical to base columns, and therefore very few changes are needed during SQL runtime to accelerate query execution. The next few subsections provide more details into how DIMEs are eventually leveraged for query acceleration.

5.1 SQL Compilation and Optimization

The SQL compiler generates a logical expression tree of operands (where an *operand* can be base columns, constants, or operators) during query parsing. By then, the compiler would have decomposed VCs into operators with base column operands. As such, the only hint that VCs were directly used in the query would be meta-data associated with operator nodes indicating a VC. No additional changes are needed during query compilation to utilize DIMEs. As an example, consider the following query:

```
SELECT UPPER(item_name)
FROM sales
WHERE category = 'household' and
price * (1-discount) > 1000;
```

Possible DIMEs are the SELECT expression `UPPER(item_name)`, the WHERE clause predicate sub-expression `(1-discount)`, and top-level expression `price * (1-discount)`. Figure 7 presents a logical expression tree for the predicate clause, with possible DIMEs highlighted.

The logical expression tree is then processed by the optimizer to generate an execution plan. The optimizer will generate an in-memory execution plan (via the table scan row source) if the cost is less than a non-in-memory plan [7]. The optimizer takes filter and decompression costs into consideration, as well as the percentage of the table being processed in-memory – recall that

with Oracle DBIM, the entire table/segment need not be in-memory [4]. Similar costing needs to be applied when dealing with DIMEs. If the table/segment is in-memory and has DIMEs stored in IMEUs, then filters that could make use of these DIMEs would cost considerably less. However, if the percentage of DIMEs in-memory is below a certain threshold, late materialization of the VC values might be cheaper, particularly if more rows will be filtered by higher-level SQL row sources (such as join), or there is a high computation cost associated with the expression.

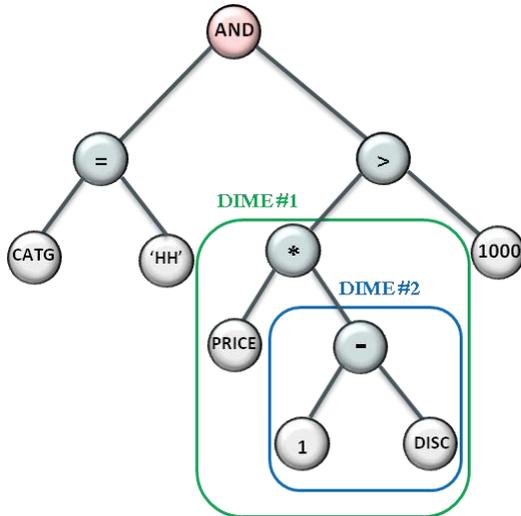


Figure 7. Logical expression tree showing top-level expression $price * (1 - discount)$ as DIME#1 and sub-expression $(1 - discount)$ as DIME#2

5.2 Code Generation for Table Scan Row Source

The table scan row source is tasked with fetching all referenced columns in a query from storage layers and applying filters before projecting passing rows to higher-level SQL row sources for further processing. The required columns are specified in a *row vector*. The row vector is just an array of operands describing base columns. For the DIME feature, the row vector was modified to support both base columns and VCs. For example, for the query in Section 5.1, the row vector would normally contain `item_name`, `category`, `price` and `discount`. With DIMEs, the row vector also contains the VCs `UPPER(item_name)`, `(1-discount)` and `price*(1-discount)`. Including VCs in the row vector allows the scan layer to project DIMEs up the query execution plan to higher-level SQL row sources which require these expressions, including *aggregation operators*, such as `SUM()` or `MIN()`, on VCs.

The scan engine constructs a runtime execution plan from the logical expression tree. First, the logical expression tree is traversed to look for expression operators that have been marked as VCs. The tree is then modified to insert branch nodes where VCs are referenced (see Figure 8). These branch nodes are needed because only at runtime is it known whether a DIME exists within the IMEU for the IMCU being processed. If there is no DIME, then processing should follow the “normal” path, which forces the expression to be computed from the base columns, essentially

reverting to the original logical expression tree. If the DIME does exist, processing will follow the optimized path in which the evaluation of the expression is folded into a reference to the DIME (which basically resembles a “base” column). Furthermore, sub-expressions may be replaced with DIME references within nested expressions. For example, it is possible for the sub-expression `(1-discount)` to be a DIME, but not the top-level expression `price * (1-discount)`. Figure 9 depicts the alternate paths chosen for evaluation when either expression is available as a DIME in an IMEU.

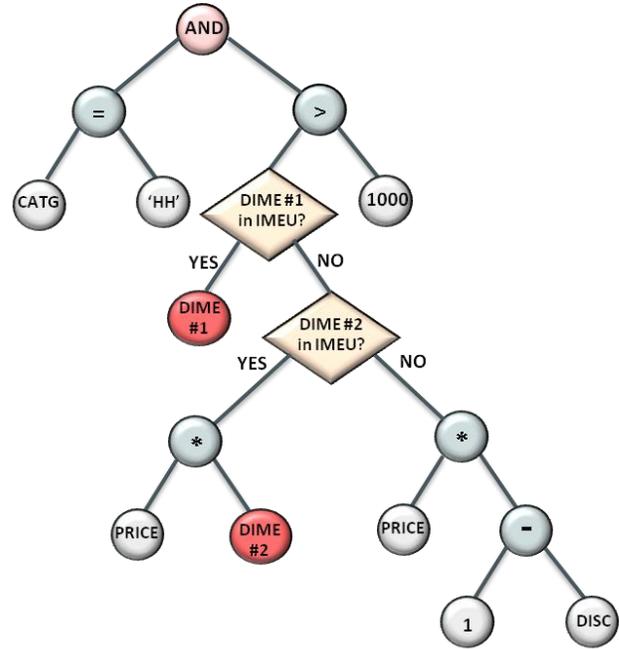


Figure 8. Modified logical-expression tree with DIME branch nodes

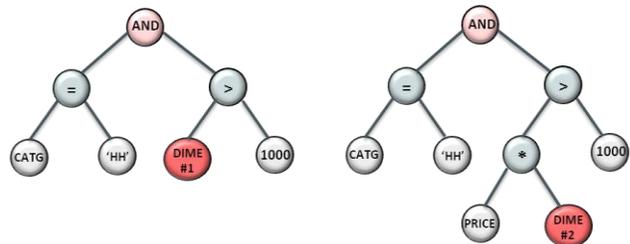


Figure 9. Run-time trees with DIME#1 or DIME#2 available in-memory in IMEUs

5.3 Execution of Table Scan Row Source

The runtime execution plan is evaluated on a per IMCU basis. Evaluation structures are first updated to point to the physical locations of the required base and virtual columns found in the corresponding IMCU/IMEU. Because IMCUs and IMEUs are stored in virtually identical formats (and their corresponding CUs/EUs share formats), gathering the required VCs from the IMEU is almost identical to gathering the required base columns from the IMCU.

Columns may or may not be present in an IMCU because with DBIM, users are allowed to specify what columns should be stored in-memory. This allows memory to be used efficiently based on the workload. Similarly, certain DIMEs may not be present in an IMEU because of space considerations. If a base column is not present in the IMCU, the scan falls back to buffer cache until the next IMCU is found. If a requested DIME is not in the IMEU, the scan executes the fallback path within the compiled expression tree, which evaluates the expression using base columns.

Because DIMEs are represented as VCs, and VCs are logically equivalent to base columns in the runtime execution plan, evaluation of DIMEs requires practically no changes in the scan engine. As such, existing DBIM scan optimizations and efficient projection techniques, such as late materialization [10], extend naturally to DIMEs.

5.3.1 Scan Engine Optimizations

The scan layer has been extensively optimized to achieve high performance for DBIM. One such optimization involves maintaining storage indexes per IMCU, where storage indexes are basically metadata describing the column values, such as minimum and maximum values. By evaluating predicates directly on storage indexes, entire IMCUs can be pruned efficiently without performing full columnar scans. Furthermore, because storage indexes are maintained separately from the columns themselves, decompression costs are saved when storage indexes are successfully applied. Since IMEUs are essentially identical to IMCUs, storage indexes are available for DIMEs as well. For example, if the predicate in a query is $price * (1 - discount) < 0$, then before the DIME is fully accessed, the minimum value from the storage index in the IMEU is checked to see if any rows will pass the predicate.

Other scan optimizations performed include utilizing SIMD instructions for fast vector processing on columnar data [3,4]. By transforming a complex multi-column expression into a VC stored as a DIME in the IMEU, all the hardware optimized techniques for columnar evaluation can be applied to DIMEs. The alternate method would involve a costly row-by-row evaluation of the expression requiring loading and processing each column operand in the expression.

Scan optimizations are also tailored to specific columnar data formats. For example, many columns in DBIM are formatted using dictionary-encoding [3]. Expression evaluation on dictionary-encoded column vectors can reduce computation and bandwidth costs considerably. For instance, consider the predicate $upper(substr(a, 1, 3)) = 'DOG'$. If a DIME exists for the sub-expression $substr(a, 1, 3)$, then the predicate is effectively transformed to $upper(DIME) = 'DOG'$. With the DIME being dictionary-encoded in the IMEU, the predicate can be efficiently evaluated on the dictionary itself [3,4].

5.3.2 Projection and Late Materialization

Projection is the process of sending passing rows up from the table scan row source to higher-level SQL row sources for further evaluation. Project expressions are described similarly to predicate expressions – i.e. via logical expression tree – so branch nodes exist as decision points in the tree which check whether the DIME result exists in-memory or not. For example, if the query contains a SELECT clause involving a DIME expression, the DIME result values are directly projected, while the underlying base columns in the expression can be safely ignored (assuming

they aren't needed by other expressions). If the DIME does not exist, then the underlying base columns are projected instead.

Virtually all project optimizations that are performed on base columns can also be performed on DIMEs. One optimization worth noting is that, for dictionary-encoded DIMEs, it is possible to return the dictionary indices of the passing rows themselves, and not the actual values. This late materialization can provide significant performance gains because a) the calling layers can sometimes operate more efficiently on these indices directly, and b) the calling layers do not always need the full symbol information if, for instance, further post-filter predicates or a join are applied [10].

6. PERFORMANCE EVALUATION

In this section, we present some experimental results to demonstrate the benefits of the DIME feature: 1) Improved response times for analytic queries, 2) Reduced CPU utilization and 3) Higher throughput for mixed workloads that combine analytics and transaction processing.

6.1 Accelerating Analytic Queries

In this section, we demonstrate the performance speed-ups achieved by analytic queries in three different experimental setups. The first experiment demonstrates the possible benefits of this feature through the use of explicitly declared in-memory VCs. The next experiment demonstrates the ability of the ESS to automatically capture frequently evaluated expressions across an analytic workload, and showcase the benefits of materializing the hottest expressions as DIMEs. The final experiment focuses on how DIMEs can improve JSON query processing by an order of magnitude. All of these experiments are conducted on an Oracle Exadata Database machine [12], which is a state-of-the-art database SMP server and storage cluster system.

6.1.1 In-Memory Virtual Columns

A 14-column, 100 million row, non-partitioned 'Atomics' table with storage size of 8GB is chosen for this experiment. The table is configured with default in-memory compression levels. Four virtual columns representing mathematical expressions and string manipulations are manually added to the table (see Table 2). The column *rand1m* contains uniformly distributed random values from 1 to 1,048,575. Similarly, columns *rand15* and *rand64k* contain uniformly distributed random values from 1 to 15, and 1 to 65,535 respectively. The column *uniq100m* contains 100 million unique values in the range 1 to 104,857,600. The column *randstringsize26* consists of uniform random strings derived from letters 'a,b,c,...z'. The entire table, including VCs, is populated into the IM column store.

Table 2. List of user-defined VCs

VC Name	Expression
VC1	$(rand64k/1000) + (rand1m/1000)$
VC2	$((1 - (rand15/100)) + (rand1m/10) + rand64k)$
VC3	$(0.3 * rand15)$
VC4	<code>SUBSTR(randstringsize26, 10, 5)</code>

Table 3 depicts a set of five analytic point queries that were run against this table. The queries have a mixture of expressions in the WHERE clause as well as inside aggregations in the SELECT list. More specifically, Q1, Q2 and Q3 have top-level expressions materialized as DIMEs in IMEUs. Q4 and Q5, however, have

only sub-expressions materialized as DIMEs. Figure 10a and 10b depict the gain in response times seen by using DIMEs versus a regular DBIM scan. All queries were run serially.

Table 3. Point queries on Atomics table

ID	QUERY
Q1	<pre>SELECT MAX(rand15) FROM atomics WHERE ((rand64k/1000)+(rand1m/1000))=10;</pre>
Q2	<pre>SELECT MAX((1-(rand15/100))+(rand1m/10)+rand64k) FROM atomics;</pre>
Q3	<pre>SELECT MAX(uniq100m) FROM atomics WHERE ((1-(rand15/100))+(rand1m/10)+rand64k)=10000;</pre>
Q4	<pre>SELECT MAX(uniq100m) FROM atomics WHERE (0.3*rand15)+((rand64k/1000)+(rand1m/1000))<100;</pre>
Q5	<pre>SELECT MAX(rand15) FROM atomics WHERE UPPER(SUBSTR(randstringsize26,10,5))='limja';</pre>

Figure 10a demonstrates that by materializing top-level expressions as DIMEs, we can achieve upwards of 1000X improvement in query response times. Figure 10b, on the other hand, shows that by materializing only sub-expressions in-memory, query response times improve by a modest factor of 2. This can be explained by the fact that when only sub-expressions are materialized in-memory, the scan still needs to perform run-time expression evaluation to obtain the top-level expression result before the predicate can be applied. Since these queries are quite short in duration, this top-level expression evaluation dominates the data processing cost, and hence, stifles the speed-up achieved by DIMEs.

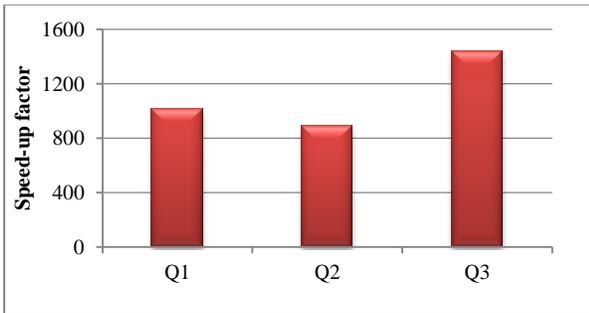


Figure 10a. Speed-up in Atomics queries with top-level expressions materialized as DIMEs

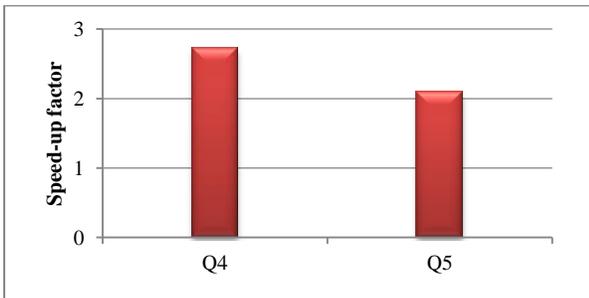


Figure 10b. Speed-up in Atomics queries with only sub-expressions materialized as DIMEs

6.1.2 Analytic Workload: Auto-capture of DIMEs and Query Acceleration

While it is true that materializing user-defined VCs is a simple technique to obtain faster query response times, choosing which expressions to create VCs on is a challenging task in its own right. An expression may be occurring frequently in the SQL queries, but may not get evaluated enough due to high filter rate of certain predicates. In addition, the query optimizer may choose to rewrite the query in such a way that expression evaluation is no longer the dominant processing cost. Moreover, a DBA has no knowledge of implicit expressions that the optimizer generates. Hence, the task of capturing expressions is best left to the ESS.

In this experiment, we use a TPC-H based analytic schema [13] with eight tables (30 GB scale factor) to test the expression tracking efficiency of the ESS. TPC-H is a decision support benchmark which consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The analytic queries in this benchmark are fairly complex, examining large amounts of data to arrive at answers to critical business questions. Table 4 lists the top 7 hottest expressions captured by the ESS from a workload comprising of several analytic queries.

Table 4. Top expressions captured by ESS

Table Name	Expression
LINEITEM	<code>l_extendedprice * (1-l_discount)</code>
LINEITEM	<code>l_extendedprice * (1-l_discount) * (1+l_tax)</code>
ORDERS	<code>CASE when (o_orderpriority<>(1-urgent) and o_orderpriority<>(2-high)) then 1 else 0 END</code>
ORDERS	<code>CASE when (o_orderpriority<>(1-urgent) and o_orderpriority<>(2-high)) then 1 else 0 END</code>
LINEITEM	<code>CASE when l_receiptdate > l_commitdate then 1 else 0 END</code>
ORDERS	<code>SYS_OP_BLOOM_FILTER(:BF0000, o_custkey)</code>
LINEITEM	<code>SYS_OP_BLOOM_FILTER(:BF0000, l_partkey)</code>

Most of the top expressions tracked by ESS are on the LINEITEM fact table. Some of these expressions are used in SELECT lists as part of aggregations while others are used in WHERE clause predicates. The ESS also tracks bloom filters, which are internal filters that are used to speed up complex joins [4,7]. While bloom filters are not directly materialized as DIMEs, the knowledge of their existence provides the SQL engine with the ability to materialize certain internal computations in IMCUs that can lead to improved join performance. A detailed discussion of these optimizations, however, is beyond the scope of this paper.

We demonstrate the power of DIMEs by considering the case when only one of these expressions: `l_extendedprice * (1-l_discount)` on the LINEITEM table, is materialized as a DIME in IMEUs. Table 5 depicts a subset of the analytic queries involving the LINEITEM table. Q1 performs filtering based on expressions in the predicate; Q2 performs an aggregation on the expression; and Q3 is a more complicated query that performs aggregations as well as grouping operations. All queries are run with a Degree of Parallelism (DOP) of 4.

Table 5. Analytic queries on LINEITEM

ID	QUERY
Q1	<pre>SELECT SUM(l_quantity) FROM lineitem WHERE (l_extendedprice*(1-l_discount))> (SELECT AVG(l_extendedprice*(1-l_discount)) FROM lineitem);</pre>
Q2	<pre>SELECT MAX(l_extendedprice*(1-l_discount)) FROM lineitem;</pre>
Q3	<pre>SELECT l_returnflag, l_linestatus, SUM(l_quantity),SUM(l_extendedprice), SUM(l_extendedprice*(1-l_discount)), SUM(l_extendedprice*(1-l_discount)*(1+l_tax)), COUNT(*) FROM lineitem GROUP BY l_returnflag, l_linestatus;</pre>

Figure 11 shows the speed-up in response times obtained by using DIMEs compared to performing vanilla in-memory scans on base columns. Q1 and Q2 benefit the most because they have the entire expression `l_extendedprice * (1-l_discount)` stored in-memory as a DIME. Q3, however, has the DIME both as a top-level expression (in `SUM(l_extendedprice*(1-l_discount))`) and as a sub-expression (in `SUM(l_extendedprice*(1-l_discount)*(1+l_tax))`). Thus, Q2 incurs the additional runtime cost of computing the second top-level expression from the DIME sub-expression, which limits the benefits seen by the DIME feature.

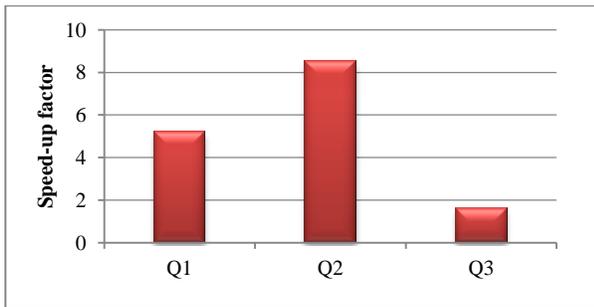


Figure 11. Speed-up in analytic queries on LINEITEM with `l_extendedprice * (1-l_discount)` materialized as a DIME

6.1.3 JSON Query Acceleration

Oracle Database 12c introduced native JSON support in 2014. Now, users can not only store JSON data in its native structure within the database, but also retrieve it in a simple JSON friendly way and access it fully transparently via SQL [15]. This gives users greater flexibility in terms of managing their JSON data within a relational database. With the DIME feature, users can improve JSON query processing by an order of magnitude. The simplest JSON expression that the ESS can capture is a `JSON_VALUE` expression that enables the user to select one top-level scalar value from within the JSON document. `JSON_VALUE` essentially acts a bridge from a JSON value to a SQL value.

For the purposes of this experiment, we focus on the NoBench benchmark suite [11]. It consists of a series of JSON objects with hierarchical data, dynamic typing and sparse attributes. The chosen JSON schema consists of 64M rows and is approximately

40GB on disk. The table is enabled for in-memory storage at default compression level. The ESS captures several `JSON_VALUE` expressions from the analytic query workload. However, we only choose 2 expressions to materialize as DIMEs, namely:

1. `JSON_VALUE(jobj, '$.num' RETURNING NUMBER)`
2. `JSON_VALUE(jobj, '$.dyn1' RETURNING NUMBER)`

Table 6 shows the set of 3 queries we choose to demonstrate improvements in JSON query processing response times. All queries run with a DOP of 32.

Table 6. JSON queries

ID	QUERY
Q1	<pre>SELECT COUNT(*) FROM nobench_main WHERE json_value(jobj, '\$.num' returning NUMBER) BETWEEN 1 AND 1000;</pre>
Q2	<pre>SELECT COUNT(*) FROM nobench_main WHERE json_value(jobj, '\$.num' returning NUMBER) BETWEEN 1 AND 100000 GROUP BY json_value(jobj, '\$.thousandth');</pre>
Q3	<pre>SELECT COUNT(*) FROM nobench_main WHERE json_value(jobj, '\$.dyn1' returning NUMBER) BETWEEN 1 AND 1000;</pre>

Queries Q1 and Q3 benefit directly from the materialized `JSON_VALUE` DIME, and achieve a 20X improvement in response times. Query Q2 needs to perform a group by on a `JSON_VALUE` expression that is not available as a DIME and hence, has a gain of only 5X (see Figure 12).

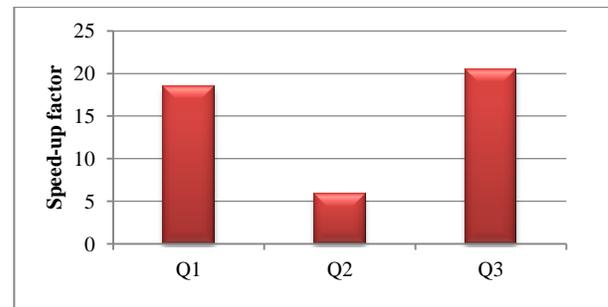


Figure 12. Performance boost in JSON query processing with `JSON_VALUE` DIMEs

6.2 Accelerate OLTAP Mixed Workloads

In modern business organizations, the ability to combine transactional processing with super-fast on-demand analytics on real time operational data is paramount to making key business decisions. Oracle DBIM is an industry-first dual format database that provides blazingly fast in-memory analytic performance while improving transactional processing. The DIME feature further strengthens DBIM performance under OLTAP mixed workloads.

This experiment comprised of a synthetic OLTAP workload that simulates an insert/update/delete workload interspersed with analytic queries. The test consists of a wide table with 6M rows, and 101 columns (1 identity column, 50 number columns and 50

varchar2 columns) with an index on the identity column. The hardware setup was a 2x Intel Xeon E5-2690 @ 2.90GHz, 8-core processor with 256GB of DRAM, of which only 60GB was used for the in-memory area. The test was run for 1 hour with all operations done with a target throughput of 2000 ops/sec. The percentage of DMLs and analytic queries in the workload was tunable – we demonstrate performance improvements for a workload with 99% DMLs and only 1% analytic scans.

We use various metrics such as query response times, CPU usage, and operation throughput (transactions or scans) to show the capabilities of the DIME feature. The analytic queries involved several expressions – the ones materialized as DIMEs are listed in Table 7. Table 8 lists a subset of the queries that were run in this workload.

Table 7. List of DIMEs materialized in-memory

ID	Expression
E1	<code>ROUND (n2 / 1000000 + n3 / 1000000)</code>
E2	<code>1 + (n2 / 1000000) + (n3 / 1000000) + (n4 / 1000000)</code>

Table 8. Analytic queries in synthetic OLTAP workload

ID	Expression
Q1	<code>SELECT MAX ((1 + (n2 / 1000000) + (n3 / 1000000) + (n4 / 1000000))) FROM c101_6p1m_hash;</code>
Q2	<code>SELECT MAX (n2) FROM c101_6p1m_hash WHERE ROUND (n2 / 1000000 + n3 / 1000000) = 10;</code>
Q3	<code>SELECT MAX (n3) FROM c101_6p1m_hash WHERE (1 + (n2 / 1000000) + (n3 / 1000000) + (n4 / 1000000)) = 8;</code>

Figure 13 shows the improvement in response times of analytic queries (Q1, Q2 and Q3) in the OLTAP workload, obtained by using DIMEs over vanilla DBIM scans. It can be seen that the median query response time improves by a factor of almost 200X. In addition, the workload can successfully sustain the target throughput rate of 2000 ops/sec, while limiting average CPU utilization to only 28.6%. In contrast, without DIMEs, not only does query performance suffer, but average CPU utilization is also at 100%, thereby not achieving the expected throughput rate.

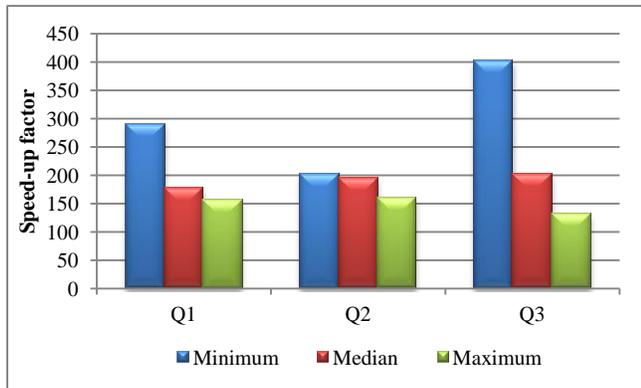


Figure 13. Speed-up in minimum, median and maximum response times of queries in OLTAP workload with DIMEs

Thus, the DIME feature guarantees excellent analytic performance while efficiently utilizing CPU and other system resources even in a mixed OLTAP workload.

7. CONCLUSIONS AND FUTURE WORK

Expression evaluation in queries is the proverbial “dark matter” of analytic workloads – invisible to most performance monitoring tools yet consuming considerable CPU cycles. This paper presents how Oracle Database 12.2 tackles this problem using a novel technique called Dynamic In-Memory Expressions (DIMEs) that greatly reduces expression evaluation cost, thereby significantly accelerating analytic queries.

DIMEs are automatically captured from the database and materialized in the IM column store without any user intervention. The DIME feature is seamlessly integrated with the Oracle Database In-Memory (DBIM) infrastructure, which allows us to apply all the in-memory query optimizations introduced for DBIM on DIMEs. We show that DIMEs can yield integral multiples of speedup (up-to 1000x) in analytic queries on relational as well as JSON schemas.

Future work includes integrating the DIME infrastructure with the Automatic Data Optimization (ADO) framework [16] to improve memory management of DIMEs by automatically migrating IMEUs between different storage tiers based on access frequency; supporting DIMEs on Active Dataguard (physical standby); augmenting the ESS to capture expressions involving columns from different tables; and extending the DIME storage to Flash and other emerging persistent memory technologies such as NVRAM.

8. ACKNOWLEDGEMENTS

We acknowledge the contributions of members in Oracle Data, Space, Transactions, SQL, JSON, Functional Testing and Stress Testing teams involved throughout the product lifecycle. We also thank the Performance team for helping us with performance evaluation of the architecture.

9. REFERENCES

- [1] Boncz, P., A., Grust, T. et. al. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2006)
- [2] Stonebraker, M., Abadi, D., et. al. C-Store: A Column-oriented DBMS. *Proceedings of the 31st VLDB Conference* (2005)
- [3] Lahiri, T. et. al. Oracle Database In-Memory: A Dual Format In-Memory Database. *Proceedings of the ICDE* (2015)
- [4] Oracle Database In-Memory, an Oracle White Paper, 2015
- [5] P. Boncz, T. Neumann, and O. Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In TPCTC, 2013
- [6] W. Bridge, A. Joshi, M. Keihl, T. Lahiri, J. Loaiza, and N. MacNaughton, The Oracle Universal Server Buffer Manager. *Proceedings of VLDB '97*, pp. 590-594, 1997.
- [7] D. Das et. al. Query optimization in Oracle 12c database in-memory. *Proceedings of VLDB, 2015*, pp. 1770-1781.
- [8] Query Optimization in Oracle Database 10g Release 2, an Oracle White Paper, 2005

- [9] Oracle Total Recall with Oracle Database 11g Release 2, an Oracle White Paper, 2009
- [10] D.J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented DBMS. *Proceedings of ICDE*, 2007.
- [11] C. Chasseur, Y. Li, and J. M. Patel. Enabling JSON Document Stores in Relational Systems. *Proceedings of WebDB*, pages 1-6, 2013.
- [12] R. Greenwald, M. Bhuller, R. Stackowiak, and M. Alam, *Achieving Extreme Performance with Oracle Exadata*, McGraw-Hill, 2011
- [13] TPC Benchmark H (Decision Support) Standard Specification Revision 2.17.1
- [14] B. Dageville, M. Zait: SQL Memory Management in Oracle 9i. *Proceedings of VLDB*, 2002
- [15] Z.H. Liu, B. Hammerschmidt, D. McMahon: JSON data management: supporting schema-less development in RDBMS. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2014): 1247-1258
- [16] Automatic Data Optimization with Oracle Database 12c, an Oracle White Paper, 2015