Oracle’s In-Memory Database Strategy for OLTP and Analytics

Tirthankar Lahiri, Markus Kissling
Oracle Corporation

Keywords: TimesTen, Database In-Memory, Oracle Database 12c

Introduction

With increasing DRAM densities, systems with hundreds of gigabytes and even terabytes of memory are becoming common. This trend is exemplified by Oracle’s SPARC M6-32 SuperCluster, an enormous system featuring up to 32 TeraBytes of main memory. A large fraction of user data, and in many cases, all user data, can thus be stored entirely in DRAM avoiding the need for costly disk IO. Oracle has two complementary In-Memory Database Technology offerings for the Application Tier and for the Database Tier, as depicted in Illustration 1 below.

1) **Oracle TimesTen**: For applications requiring ultra-low response time, Oracle TimesTen is a specialized, memory-resident relational database that can be directly embedded within an application, achieving microsecond response times. TimesTen can be deployed either as a standalone database, or as a high performance fully transactional cache of data sourced from an underlying Oracle Database. TimesTen is therefore typically deployed within the Application-Tier, and its primary purpose is to provide ultra-low latency for custom OLTP applications.

2) **Database In-Memory**: Within the Database Tier, the Oracle Database In-Memory option features a unique dual-format architecture enabling orders of magnitude faster Analytic workloads (Billions of rows per second processing speeds), as well as significantly faster Enterprise workloads such as CRM, HCM, ERP through the elimination of analytic indexes. With Database In-Memory, the total size of the database is not limited by the size of main memory: The in-memory column store can be used for important tables, while the buffer cache and smart flash cache (on Exadata storage) can be used to provide tiered storage across memory, flash, and disk.

Illustration 1: Cross-Tier In-Memory Database Technology
In this two part paper, we first provide an overview of the TimesTen In-Memory Database and then conclude with an overview of Oracle Database In-Memory.

Part 1: Oracle TimesTen In-Memory Database

1.1 Architectural Overview of TimesTen

The Oracle TimesTen In-Memory database is a memory-resident and completely persistent ACID compliant database. As shown in Illustration 2 below, TimesTen features extremely high performance as a result of its fully memory-based design: The entire database is in-memory at all times and can be accessed in-process, without any network latency. The stunning performance of TimesTen is the reason it is deployed by thousands of customers worldwide in applications requiring real-time responsiveness: Financial Services, Telecommunications, Network Management, Ecommerce and Online Retail, to name a few of TimesTen’s many customer segments.

1.1.1 Overview: APIs and Connectivity

TimesTen provides standard APIs for database operations. These include ODBC, JDBC, as well as the Oracle Call Interface (OCI). Applications connect to TimesTen either via a conventional client-server protocol, or an optimized direct-link protocol in which the application directly links the TimesTen library. In direct-link mode, all database API invocations are treated simply as function calls into the TimesTen shared library allowing for in-process execution of database code.

1.1.2 Overview: Basic Storage Manager Design

Some basic design principles guide the design of the storage manager. Concurrency control mechanisms are both light-weight (e.g. using lightweight latches instead of database locks whenever possible) and fine-grained, to allow for maximal scaling on multi-core architectures (as demonstrated by the throughput shown in Illustration 2 above).

Another over-arching design principle is the use of memory-based addressing rather than logical addressing. For instance, indexes in TimesTen contain pointers to the tuples in the base table. The metadata describing the layout of a table contains pointers to the pages comprising the table. Therefore both index scans and tablescons can operate via pointer traversal. This design approach is repeated over and over again in the storage manager, with the result that TimesTen is significantly faster than a disk-oriented database even one that is completely cached – since there is no overhead from having to translate logical rowids to physical memory addresses of buffers in a buffer cache.

Multiple types of indexes are supported by TimesTen. TimesTen supports Hash Indexes for speeding up lookup queries, Bitmap Indexes for accelerating star joins with complex predicates, as well as Range Indexes for accelerating range scans. Since the table data is always in memory, index leaf nodes do not need to store key values, with significant space savings.
1.1.3 Overview: Transactional Durability

TimesTen has been designed from the ground up to have full ACID properties. TimesTen employs checkpointing with write-ahead logging for durability. Logging is frequently a scaling bottleneck on large multi-core architectures, and for this purpose – again in keeping with the fine-grained concurrency control design motif, TimesTen has a multi-threaded logging mechanism so that generation of log is not serialized. In this mechanism, the TimesTen log buffer is divided into multiple partitions which can be populated in parallel by concurrent processes. Sequential ordering of the log is restored when the log is read from disk. For the highest performance, TimesTen offers an optional delayed-durability mode: an application commits by writing a commit record into the log buffer without waiting for the log to be forced to disk. The log is periodically flushed to disk every 200 mSec as a background activity. This mode of operation allows the highest possible throughput with the possibility of a small (bounded) amount of data loss, unless the system is configured with 2-safe replication as described later.

The database has 2 checkpoint files for the permanent data, and a variable number of log files. Once a database checkpoint is completed, the checkpointing operation switches to the other file. With this approach, one of the checkpoint files is always a completed checkpoint of the in-memory contents and can be used for backup and roll-forward recovery. TimesTen supports fuzzy checkpointing with user configurable rate controls.

TimesTen offers read-committed transactional isolation by default (serializable isolation as an application choice). With read-committed isolation, application bottlenecks are minimized since TimesTen has a lockless multi-versioning mechanism for read-write concurrency and row-level locking for maximal write-write concurrency. Row-level multi-versioning allows readers to avoid getting any locks altogether, while row-level locking provides the highest possible scalability for update intensive workloads.

1.2 TimesTen High Availability

The vast majority of TimesTen deployments use replication for high-availability. TimesTen replication is log-based, relying on shipping log records for committed transactions from a transmitter database to a receiver database, on which the changes in the log are then applied. Replication can be configured with multiple levels of resiliency. For the highest resiliency, TimesTen provides 2-safe replication. With 2-safe replication, a transaction is committed locally only after the commit has been successfully acknowledged by the receiver. 2-safe replication is often used with non-durable commit. This combination allows applications to achieve commit durability in two memories, without requiring any disk IO.

Illustration 5: TimesTen Replication
Replication can be at the level of individual tables or at the level of the whole database. Replication can be configured to be multi-master and bi-directional. The preferred configuration for replication is to have whole database replication from an Active Database to a Standby database. Standby databases can in turn propagate changes to a number of Read-Only Subscriber databases, as depicted in Illustration 3. In this type of replication scheme, the active can host applications that both read and modify the database. The standby and the read-only subscriber databases can host read-only applications. If a failure of the Active occurs, the Standby can be promoted to an Active database. If a failure of the Standby occurs, the Active will then replicate directly to the Read Only Subscribers.

In keeping with the design principle of end to end parallelism, replication can be parallelized across multiple replication tracks. Each track consists of a replication transmitter on the sending side, and a replication receiver on the receiving side. Parallel replication preserves commit ordering, but allows transactions without any data dependencies to be applied in parallel by the receiver. Together with log parallelism, replication parallelism provides end to end parallelism between the master and the receiver, and the highest possible throughput.

1.3 Application-Tier Cache Grid

The Application-Tier Cache Grid configuration (shown above in Illustration 4) allows TimesTen to be deployed as a persistent transactional cache for data in an Oracle database. Deploying TimesTen as an application-tier cache can greatly accelerate an application even when the database working set is fully cached in memory within the buffer cache. This is for two reasons:

1) Application Proximity: The TimesTen database can be collocated in the application-tier, resulting in lower communication costs to the database for access to cached data. For the best response time, TimesTen can be directly linked to the application providing in-process execution of operations on cached data.

2) In-Memory Optimizations: As stated earlier, the TimesTen storage manager is built assuming full memory residency, it requires far fewer instructions for database operations than disk-based databases.
TimesTen allows a collection of tables to be declared as a Cache Group against corresponding tables on an Oracle database. A cache group is a syntactic declaration and comprises one Root table, and optional Child tables related by foreign key constraints. For instance, for an ecommerce application, one approach to creating a cache group would be to have a root table for all user profiles, and child tables for open orders placed by each user, as well as a list of service requests for each user. This cache group is depicted in Illustration 5 below.

Illustration 5: Example Cache Group

Cache groups can be loaded from an Oracle database in one of two ways:

1) **Pre-loaded**: In this mode, the cache group is explicitly loaded before the workload is run. For instance in the above example, all users could be preloaded from the Oracle RDBMS into the cache group.

2) **Dynamically loaded**: In this mode, data is brought into the cache when referenced. For instance, when a user logs in, it could be at that point that the user profile record for the user and all the associated Orders and Preference records are brought into the cache group so that all subsequent references by that user will benefit from in-memory processing (with a penalty on the first reference only). The data to be referenced must be identified by an equality predicate on the primary key of the root table (in this case, by the user profile Id). The root table row and related child table rows are referred to as a cache instance.

For dynamically loaded cache groups the user can set one of two different aging policies to remove data to ensure that the database does not run out of space. TimesTen supports time-based aging (age on the value of a timestamp column) or LRU aging, based on recency of usage. When the aging mechanism removes rows from a cache group, it removes them in units of cache instances. Cache instances form the unit of caching since they are atomic units for cache replacement.

Cache groups can be of two basic types in order to handle a variety of caching scenarios. There are two corresponding data synchronization mechanisms for these cache group types.

1) **Read-Only Cache Groups**: For data that is infrequently updated, but widely read, a read-only cache group can be created on TimesTen to offload the backend Oracle database. Very hot reference data, as online catalogs, airline gate arrival/departure information, etc. is a candidate for this type of caching. The Oracle side tables corresponding to Read-Only cache groups are updated on Oracle. The updates are periodically refreshed into TimesTen using an automatic refresh mechanism.

2) **Updatable Cache Groups**: For frequently updated data, an updatable cache group with write-through synchronization is appropriate. Account balance information for an online ecommerce application, the location of subscribers in a cellular network, streaming sensor data, etc. are all
candidates for write-through caching. TimesTen provides a number of alternative mechanisms for propagating writes to Oracle, but the most commonly used and highest performing mechanism is referred to as Asynchronous Writethrough where the changes are replicated to Oracle using a log-based transport mechanism. This mechanism is also capable of applying changes to Oracle in parallel, in keeping with the parallel-everywhere design theme of the system.

It is possible to deploy multiple TimesTen cache databases against a single Oracle database. This architecture is referred to as the Application-Tier Database Cache Grid (depicted in Illustration 5 above). Cache groups can once again be classified into two categories of data visibility on a grid.

1) **Local Cache Groups**: The contents of a local cache group are not shared and are visible only to the grid member they are defined on. This type of cache group is useful when the data can be statically partitioned across grid members; for instance, different ranges of user profile Ids may be cached on different grid members.

2) **Global Cache Groups**: In many cases, an application cannot be statically partitioned and Global Cache Groups allow applications to transparently share cached contents across a grid of independent TimesTen databases. With this type of cache group, cache instances are migrated across the grid on reference. Only consistent (committed) changes are propagated across the grid. In our example, a user profile and the related records can be loaded into one grid member on initial login by the user. When the user disconnects and later logs in onto a different grid member, the records for the user profile and related child table entries are migrated from the first grid member to the second grid member. Thus, the contents of the global cache group are accessible from any location, via data shipping.

The cache grid also provides data sharing via function shipping or global query. A global query is a query executed in parallel across multiple grid members. For example, if a global query needs to compute a COUNT(*) on the user profiles table, it would ship the count operation to all the grid members, and then collate the results by summing the received counts. This mechanism can be generalized to much more complex queries involving joins, aggregations, and groupings. Global queries are useful for running reporting operations on a grid, for example, what is the average value of the current outstanding orders for all users on the grid.
Part II: Oracle Database In-Memory

2.1 Overview of Oracle Database In-Memory

The Oracle Database In-Memory option provides a unique dual-format row/column in-memory representation thus avoiding the tradeoffs inherent in single-format in-memory databases. Unlike traditional in-memory databases, the new In-Memory option does not limit the size of the database to the size of available DRAM: the numerous optimizations at all levels in the storage hierarchy: DRAM, Flash, Disk, as well across machines in a RAC cluster, continue to allow databases of virtually unlimited size.

The key novelties in the approach taken with the Oracle Database In-Memory feature are as follows:

1) **Dual format**: It is well known that columnar data format is well suited for analytics since analytic queries typically access a small number of columns in a large number of rows. On the other hand, row-storage is better for OLTP workloads that typically access a large number of columns in a small number of rows. This is why row-storage is used by most OLTP databases, including present-generation in-memory OLTP databases such as Oracle TimesTen. Since neither format is optimal for all workloads, Oracle Database In-Memory features a Dual Format in-memory representation (Figure 1) with data simultaneously accessible in an In-Memory column store (IM column store) as well as in an in-memory row store (the buffer cache) as depicted in Illustration 6.

![Illustration 6: Dual-Format Database Architecture](image)

Note that this dual format representation does not double memory requirements. The Oracle Database buffer cache has been highly optimized over decades, and achieves extremely high hit-rates even with a very small size compared to the database size. Also, since the IM column store can replace analytic indexes, the buffer cache size can be further reduced. We therefore expect customers to be able to use 80% or more of their available memory for the IM column store.

2) **Unlimited Capacity**: The entire database is not required to have to fit in the IM column store since it is pointless to waste memory storing data that is infrequently accessed. It is possible to selectively designate important tables or partitions (for instance, the most recent months of orders in an orders table spanning several years) to be resident in the IM column store, while using the buffer cache for the remainder of the table. With Engineered Systems such as Exadata, the data can therefore span multiple storage tiers: high capacity disk drives, PCI Flash cache with extremely high IOPS, and DRAM with the lowest possible response time.

3) **Complete Transparency**: The IM column store is built into the data access layer within the database engine and presents the optimizer with an alternate execution method for extremely fast table scans. The only impact to the optimizer and query processor is a different cost function for in-memory scans. By building the column store into the database engine, we ensure that all of the
enterprise features of Oracle Database such as database recovery, disaster recovery, backup, replication, storage mirroring, and node clustering work transparently with the IM column store enabled.

2.2 New In-Memory Columnar Format

The new column format is a pure in-memory format, introduced in Oracle Database 12.1.0.2. The use of the new in-memory column format has no effect on the existing persistent format for Oracle Database. Because the column store is in-memory only, maintenance of the column store in the presence of DML changes is very efficient, requiring only lightweight in-memory data manipulation. The size of the IM column store is specified by the initialization parameter INMEMORY_SIZE.

2.2.1 Populating the In-Memory Column Store

The contents of the IM column store are built from the persistent contents within the row store, a mechanism referred to as populate. It is possible to selectively populate the IM column store with a chosen subset of the database. The new INMEMORY clause for CREATE / ALTER statements is used to indicate that the corresponding object is a candidate to be populated into the IM column store. Correspondingly, an object may be removed from the column store using the NO INMEMORY clause. The INMEMORY declaration can be performed for multiple classes of objects:

- Entire Tablespace
- Entire Table
- Entire Partition of a table
- Only a specific Sub-Partition within a Partition

Some examples of the INMEMORY clause are listed below in Illustration 7. Note from the last example that it is also possible to exclude one or more columns in an object from being populated in-memory. In the example, the photo column is excluded since it is not needed by the analytical workload, and having it in memory would waste memory.

```
ALTER TABLE sales INMEMORY;
ALTER TABLE revenue NO INMEMORY;
CREATE TABLE customers ...
PARTITION BY LIST
  (PARTITION p1 ... INMEMORY,
   (PARTITION p2 ... NO INMEMORY);

ALTER TABLE accounts INMEMORY
NO INMEMORY (photo);
```

Illustration 7: Example uses of the INMEMORY clause

The IM column store is populated using a pool of background server processes, the size of which is controlled by the INMEMORY_MAX_POPULATE_SERVERS initialization parameter. If unspecified, the value of this parameter defaults to half the number of CPU cores.
Note that there is no application downtime while a table is being populated since it continues to be accessible via the buffer cache. In contrast, most other in-memory databases require that applications must wait till all objects are completely brought into memory before processing any user or application requests.

Since some objects may be more important to the application than others, it is possible to control the order in which objects are populated using the PRIORITY subclause. By default, in-memory objects are assigned a priority of NONE, indicating that the object is only populated when it is first scanned. Note that the PRIORITY setting only affects the order in which objects are populated, not the speed of population. The different priority levels are enumerated in Table 1 below.

<table>
<thead>
<tr>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRITICAL</td>
<td>Object is populated immediately after the database is opened.</td>
</tr>
<tr>
<td>HIGH</td>
<td>Object is populated after all CRITICAL objects have been populated</td>
</tr>
<tr>
<td>MEDIUM</td>
<td>Object is populated after all CRITICAL and HIGH objects have been populated</td>
</tr>
<tr>
<td>LOW</td>
<td>Object is populated after all CRITICAL, HIGH, and MEDIUM objects have been populated</td>
</tr>
<tr>
<td>NONE</td>
<td>Object is only populated after it is scanned for the first time (Default)</td>
</tr>
</tbody>
</table>

*Table 1: Different Priority Levels*

2.2.2 In-Memory Compression

Each table selected for in-memory storage is populated into the IM column store in contiguously allocated units referred to as In-Memory Compression Units (IMCUs). The target size for an IMCU is a large number of rows, e.g. half a million.

Within the IMCU, each column is stored contiguously as a column Compression Unit (CU). The column vector itself is compressed with user selectable compression levels, depending on the expected use case. The selection is controlled by a MEMCOMPRESS subclause (The clause NO MEMCOMPRESS can be used if no compression is desired). There are basically three primary classes of compression algorithms optimized for different criteria:

1) **FOR DML:** This level of compression performs minimal compression in order to optimize the performance of DML intensive workloads, at the expense of some query performance.
2) **FOR QUERY:** These schemes provide the fastest performance for queries while providing a 2x-10x level of compression compared to on-disk. These schemes allow data to be accessed in-place without any decompression and no additional run-time memory.
3) **FOR CAPACITY:** These schemes trade some performance for higher compression ratios. They require that the data be decompressed into run-time memory buffers before it can be queried. The decompression imposes a slight performance penalty but provides compression factors of 5x – 30x. The CAPACITY LOW sublevel uses an Oracle custom Zip algorithm (OZIP) that provides
very fast decompression speeds; many times faster than LZO (which is the typical standard for fast decompression). OZIP is also implemented in a special-purpose coprocessor in the SPARC M7 processor. Compression levels are enumerated in Table II in ascending order of compression.

<table>
<thead>
<tr>
<th>MEMCOMPRESS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DML</td>
<td>Minimal compression optimized for DML performance</td>
</tr>
<tr>
<td>QUERY LOW</td>
<td>Optimized for the fastest query performance (default)</td>
</tr>
<tr>
<td>QUERY HIGH</td>
<td>Also optimized for query performance, but with some more emphasis on space saving</td>
</tr>
<tr>
<td>CAPACITY LOW</td>
<td>Balanced between query performance and space saving (uses OZIP)</td>
</tr>
<tr>
<td>CAPACITY HIGH</td>
<td>Optimized for space saving</td>
</tr>
</tbody>
</table>

Table II: MEMCOMPRESS levels for inmemory objects

It is possible to use different compression levels for different partitions within the same table. For example in a SALES table range partitioned on time by week, the current week’s partition is probably experiencing a high volume of updates and inserts, and it would be best compressed FOR DML. On the other hand, earlier partitions up to a year ago are probably used intensively for reporting purposes (including year-over-year comparisons) and can be compressed FOR QUERY. Beyond the one year horizon, even earlier partitions are probably queried much less frequently and should be compressed for CAPACITY to maximize the amount of data that can be kept in memory.

2.3 In-Memory Scans
Scans against the column store are optimized using vector processing (SIMD) instructions which can process multiple operands in a single CPU instruction. For instance, finding the occurrences of a value in a set of values as in Illustration 8 below, adding successive values as part of an aggregation operation, etc., can all be vectorized down to one or two instructions.

Illustration 8: SIMD vector processing
Efforts are made to optimize the use of memory bandwidth by using bit-packed formats to represent the column vectors. As a result, a large number of column values can be stored within a CPU cache line.

A further reduction in the amount of data accessed is possible due to the In-Memory Storage Indexes that are automatically created and maintained on each of the columns in the IM column store. Storage Indexes allow data pruning to occur based on the filter predicates supplied in a SQL statement. An In-Memory Storage Index keeps track of minimum and maximum values for each column CU. When a query specifies a \textbf{WHERE} clause predicate, the In-Memory Storage Index on the referenced column is examined to determine if any entries with the specified column value exist in each CU by comparing the specified value(s) to the minimum and maximum values maintained in the Storage Index. If the value is outside the minimum and maximum range for a CU, the scan of that CU is avoided.

For equality, in-list, and some range predicates an additional level of data pruning is possible via the metadata dictionary when dictionary-based compression is used. The metadata dictionary contains a list of the distinct values for each column within each IMCU. Thus dictionary based pruning allows Oracle Database to determine if the value being searched for actually exists within an IMCU, ensuring only the necessary IMCUs are scanned. All of these optimizations combine to provide scan rates exceeding billions of rows per second per CPU core.

2.4 In-Memory Joins
Apart from accelerating scans, the IM column store also provides substantial performance benefits for joins. Typical star joins between fact and dimension tables can be reduced to a filtered scan on a dimension table to build a compact Bloom filter of qualifying dimension keys (Illustration 9), and then a subsequent scan on the fact table using the Bloom filter to eliminate values that will not be join candidates. The Bloom filter thus greatly reduces the number of values processed by the join. While this method is not specific to in-memory, the optimizer is now able to select this method more frequently due to the massive reduction in the underlying table scan costs.

Illustration 9: Bloom filter based inmemory join

2.5 In-Memory Aggregations and Groupings
The basic primitives of very fast scans and joins can be further leveraged to accelerate complex reports featuring multiple levels of aggregations and groupings. A new optimizer transformation, called \textit{Vector Group By}, is used to compute multi-dimensional aggregates in real-time. The Vector Group By transformation is a two-part process similar to the well known star transformation.

Let’s consider the following query as an example, which lists the total revenue from sales of footwear products in outlet stores, grouped by store and by product:
Select Stores.id, Products.id, sum(Sales.revenue)
From Sales, Stores, Products
Where Sales.store_id = Stores.id
And Sales.product_id = Products.id
And Stores.type = “Outlet”
And Products.type = “Footwear”
Group by Stores.id, Products.id;

The execution steps (depicted in Illustration 10) are as follows:

**Step 1:** The Query begins by scanning the STORES and PRODUCTS dimensions.

**Step 2:** A new data structure called a *Key Vector* is created based on the results of each of these scans. A key vector maps a qualifying dimension key to a dense grouping key (e.g. each store that is of type “outlet” will have a non-zero grouping key numbered 1..N, while each product that is of type “footwear” will have a non-zero grouping key numbered 1..M)

**Step 3:** The key vectors are then used to create an additional multi-dimensional array known as an *In-Memory Accumulator*. Each dimension of the In-Memory accumulator will have as many entries as the number of non-zero grouping keys corresponding to that dimension; in the above example, it will be an (NxM) array.

**Step 4:** The second part of the execution plan is the scan of the SALES table and the application of the key vectors. For each entry in the SALES table that matches the join conditions based on the key vector entries for the dimension values (i.e. has a non-zero grouping key for each dimension), the corresponding sales amount will be added to the appropriate cell in the In-Memory Accumulator. At the end, the In-Memory Accumulator will have the results of this aggregation.

The combination of these two phases dramatically improves the efficiency of a multiple table join with complex aggregations. Again, this is not strictly an in-memory specific optimization, but one that can be frequently chosen by the optimizer in view of the massive speedup of scans.

*Illustration 10: Vector Group By example*
2.6 In-Memory Column Store and Full Transactional Consistency

The default isolation level provided by Oracle Database is known as *Consistent Read*. With Consistent Read, every transaction in the database is associated with a monotonically increasing timestamp referred to as a *System Change Number* (SCN). A multi-versioning protocol is employed by the buffer cache to ensure that a given transaction or query only sees changes made by transactions with older SCNs.

The IM column store similarly maintains the same consistent read semantics as the buffer cache. Each IMCU is marked with the SCN of the time of its creation. An associated metadata area, known as a Snapshot Metadata Unit (SMU) tracks changes to the rows within the IMCU made beyond that SCN.

Illustration 11: Change tracking and repopulate

The SMU tracks the validity of rows within the IMC. Changes made by transactions are processed as usual via the Buffer Cache, but for tables populated into the IM column store, the changes made to them are also logged in an in-memory “transaction journal” within the SMU.

When a column scan runs against the base IMCU, it fetches the changed column values from the journal for all the rows that have been modified within the IMCU at an SCN earlier than the SCN of the scan (according to Consistent Read semantics, changes made at an SCN higher than the SCN of the scan are not visible to the scan).

As changes accumulate in the journal, retrieving data from the transaction journal causes increased overhead for scans. Therefore, a *background repopulate* mechanism periodically rebuilds the IMCU at a new SCN, a process depicted in Illustration 11 above. There are two basic kinds of repopulate:

1) **Threshold Repopulate**: Threshold repopulate is a heuristic driven mechanism that repopulates an IMCU once the changes to the IMCU exceed a certain threshold and various other criteria are met.

2) **Trickle Repopulate**: Trickle repopulate runs constantly and unobtrusively in the background, consuming a small fraction of the available repopulate server processes. The goal of trickle repopulate is to ensure that eventually any given IMCU will be completely clean even if it is not been sufficiently modified to qualify for threshold repopulate. The number of populate background processes that can be used for trickle repopulate is controlled by another initialization
parameter: INMEMORY_TRICKLE_REPOPULATE_SERVERS_PERCENT. This parameter defaults to 1. For example, if INMEMORY_MAX_POPULATE_SERVERS = 10, at most one-tenth of a single CPU core is dedicated to trickle repopulate by default.

2.7 In-Memory Column Store Scale-Out
Apart from providing in-memory columnar storage on a single machine, Oracle Database In-Memory can scale out across multiple machines using the Oracle Real Application Cluster (RAC) configuration.

![Illustration 12: In-Memory Scale-Out](image)

On a RAC cluster, queries on in-memory tables are automatically parallelized across multiple instances as depicted in Illustration 12 above. This results in each parallel query process accessing local in-memory columnar data on each instance, with each process then only having to perform a fraction of the work.

2.7.1 Scale-Out Distribution and Parallel Execution
The user can specify how the data within a table is to be distributed across the instances of a RAC cluster by using the `DISTRIBUTE` clause. There are four possible options for thus distributing the contents of a table:

1) **DISTRIBUTE BY PARTITION**: When a table is partitioned, this distribution scheme assigns all data from the same partition to the same instance and maps different partitions to different instances. This type of distribution is specially suited to hash partitions which usually exhibit uniform access across partitions. Distributing by partition has another potential benefit: It can allow in-memory partition-wise joins between two tables that are partitioned on the same attribute. For instance, if a `SALES` table is hash partitioned on `order_id`, and a `SHIPMENTS` table is likewise hash partitioned by `order_id`, the `SALES` to `SHIPMENTS` join can be accomplished as a set of local in-memory joins on each instance since the partitions will be co-located.

2) **DISTRIBUTE BY SUBPARTITION**: In Oracle Database, tables can be composite partitioned: i.e. Partitioned by some criteria, and then each partition further sub-partitioned by a different criteria. Distributing by sub-partition can be helpful when the top-level partitioning criteria could cause skewed data access. For instance if a `SALES` table is partitioned by weekly date ranges, it is likely that distributing by partition would cause skewed access, since more recent date ranges would probably exhibit far more activity. However, if the table were further sub-partitioned by hash on `order_id`, distributing by sub-partition would provide uniform accesses across instances. Furthermore the above join between `SALES` and `SHIPMENTS` could still be done as in-memory local joins, since all the `SALES` sub-partitions will be collocated with all the `SHIPMENTS` partitions with the same `order_id` hash (another example of partition-wise join).
3) **Distribute by Rowid Range**: When a table is not partitioned, or the partitioning criteria used would cause severely skewed access, then it is possible to ensure that the table contents are uniformly distributed across the instances using **ROWID RANGE** based distribution – which places an IMCU on an instance by applying a uniform hash function to the rowid of the first row within that IMCU. This scheme therefore distributes the contents of the table without regard to the actual values. While this does result in uniform distribution of the table contents and uniform accesses across the instances, it does have the drawback of precluding partition-wise joins.

4) **Distribute Auto**: Choosing this scheme (which is the default distribution mode) indicates that it is up to the system to automatically select the best distribution scheme from the above three, based on the table’s partitioning criteria and optimizer statistics.

### 2.7.2 Scale-Out In-Memory Fault Tolerance

On Oracle Engineered systems such as Exadata and SPARC Supercluster, a fault-tolerance protocol specially optimized for high-speed networking (direct-to-wire Infiniband) is used to allow in-memory fault tolerance as depicted in Illustration 12 below.

Fault-tolerance is specified by the **DUPLICATE** subclause. For a table, partition, or sub-partition marked as **DUPLICATE**, all IMCUs for that object are populated in two instances. Having a second copy ensures that there is no downtime if an instance fails since the second copy is instantly available. Both IMCUs are master copies that are populated and maintained independently, and both can be used at all times for query processing.

![Illustration 13: In-memory column store fault tolerance](image)

For small tables containing just a few IMCUs (such as small dimension tables that frequently participate in joins), it is advantageous to populate them on every instance, in order to ensure that all queries obtain local access to them at all times. This full duplication across all instances is achieved using the **DUPLICATE ALL** subclause.

### 2.8 Conclusions and Future Work

While column oriented storage is known to provide substantial speedup for Analytics workloads, it is also unsuited to OLTP workloads characterized by high update volumes and selective queries. The Oracle Database In-Memory Option provides a true dual-format in-memory approach that is seamlessly built into the Oracle Data Access Layer, which allows it to be instantly compatible with all of the rich functionality of the Oracle Database. In-Memory storage can be selectively applied to
important tables and to recent partitions, while leveraging the mature buffer cache, flash storage and
disk for increasingly colder data.

As a result of this balanced approach, Oracle Database In-Memory allows the use of in-memory
technology without any of the compromises or tradeoffs inherent in many existing in-memory
databases. Ongoing work includes integrating Oracle Database In-Memory into the Automatic Data
Optimization (ADO) framework (introduced with Oracle Database 12c) that automatically migrates
objects between different storage tiers based on access frequency, the ability to create the in-memory
column store on an Active Dataguard (physical standby) Database (in the 12.1.0.2 release it is
supported on Logical Standby databases but not on a Physical Standby), and extending the in-memory
columnar format to Flash and other emerging persistent memory technologies.

Contact address:

Name
Tirthankar Lahiri
400 Oracle Parkway, 4OP14
Redwood Shores, CA 94065
USA
Phone: +1(650) 506 6279
Fax: +1(650) 506 6279
Email: tirthankar.lahiri@oracle.com

Markus Kissling
Liebknechtstrasse 35
Stuttgart, 70565
DE
Phone: +49 711 72840 134
Fax: +49 711 72840 134
Email: markus.kissling@oracle.com