Automatic Indexing in Oracle

Sunil Chakkappen
Oracle America Inc.
Redwood City, CA, USA
sunil.chakkappen@oracle.com

Shreya Kunjibettu
Oracle America Inc.
Redwood City, CA, USA
shreya.kunjibettu@oracle.com

Daniel McGreer
Oracle America Inc.
Redwood City, CA, USA
daniel.mcgreer@oracle.com

Masoomeh Javidi Kishi
Oracle America Inc.
Redwood City, CA, USA
masoomeh.javidi.kishi@oracle.com

Hong Su
Oracle America Inc.
Redwood City, CA, USA
hong.su@oracle.com

Mohamed Ziauddin
Oracle America Inc.
Redwood City, CA, USA
mohamed.ziauddin@oracle.com

Mohamed Zait
Databricks Inc.
San Francisco, CA, USA
mohamed.zait@gmail.com

Zhan Li
Meta Platforms Inc.
Menlo Park, CA, USA
zhanl@meta.com

Yuying Zhang
Google Inc.
Mountain View, CA, USA
yuyingzhang@google.com

ABSTRACT

Indexes are one of the important access structures that help improve database performance. This paper provides a methodology to automate the entire lifecycle of index creation and management with continuous index tuning based on changing data and workload. We present novel ideas that are critical to ensuring automatic indexing seamlessly works in a production database. Our methodology avoids using an expensive clone; yet offers non-intrusive index operations (candidate isolation and evaluation with Oracle resource manager ensuring no visible impact to the user workload), and upon deployment of auto indexes ensures non-disruptive plan invalidations and timely mitigation of performance regressions. The proposed approach is unique in that it is incremental and iterative, continually creating beneficial indexes and dropping unused ones as the workload evolves. The approach even supports indexes on expressions. It performs careful validation – including computing overhead of index maintenance incurred during DML while evaluating potential benefit – and provides accountability for its actions. Performance regressions are effectively managed using Oracle's powerful SQL Plan Management (SPM) framework. For example, a new automatic index isn't dropped in response to a single statement regressing due to it; SPM instead ensures such regressing statements revert to well-performing plans even in the presence of new indexes that continue to benefit other statements. We also share results of comprehensively evaluating various automatic indexing aspects in publicly available and Oracle customer workloads. Our experiments show benefit with automatic indexing, especially in customer workload, with a 15% improvement in performance and 60% space reclamation potential. This automatic indexing feature is available since Oracle 19c and in Oracle Autonomous Database.

PVLDB Reference Format:
Sunil Chakkappen, Shreya Kunjibettu, Daniel McGreer, Masoomeh Javidi Kishi, Hong Su, Mohamed Ziauddin, Mohamed Zait, Zhan Li, and Yuying Zhang. Automatic Indexing in Oracle. PVLDB, 18(12): 4924 - 4937, 2025.
doi:10.14778/3750601.3750616

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750616

1 INTRODUCTION

Indexes remain a critical feature for applications today on large data sets running millions of SQL statements per day. Using the right index mix helps minimize the resource utilization (CPU and IO) when fetching a relatively small amount of data from very large tables and increases the application response times and throughput. The task of creating indexes requires intimate knowledge of the data model, schema design, data distribution and application logic, as well as some knowledge of the internals of the database system on top of which the application is implemented (e.g. query optimization, buffer cache management, etc.). No matter how skilled the people who do the tuning are, they rarely revise the choice of indexes when changes are made to the data model, application code, or data distribution because the entire process is labor intensive. Not only does the lack of follow-up lead to cases of missed chances to improve system performance, but also can have a negative effect where indexes become a burden (e.g. maintenance overhead) without benefit. Furthermore, application customizations might use generic columns (known as "flex" columns) that are not commonly used across all customers of the application. They do not have indexes deployed by default from the application vendor, so it is each customer's responsibility to manually create them. Automatic index management helps to avoid these issues.

1.1 Automatic Indexing in Production

Automatic index management should be a full-fledged function of any modern database system, considering its complexity and potential impact. Many factors need to be addressed to facilitate automatic application of indexes across real-world scenarios. It needs to be aware of essential and relevant database status (e.g. existence of other physical structures and the currently available resources). Indexes that can potentially improve SQL statements' performance (candidate indexes) must be validated to see if they indeed improve

customer workloads. Physical structure advisors [11, 12, 13, 23], such as automatic indexing, clustering, and partitioning are traditionally first analyzed on a test system/clone. This is to minimize any negative impact on the production system when those physical structures are being evaluated for their benefits. Only when the test workload exhibits overall performance improvements do the new physical structures get deployed on the production system. Although safe for the production system, this approach suffers from two problems: 1) the clone must have the exact same data and access structures as the production. The storage cost of maintaining a clone, especially on large database systems, can be exorbitant. The computing cost can be also very high if the clone must constantly be synchronized with the production system. 2) It is technically challenging to capture the production workload (DMLs and queries) and replay it on the clone. Typically, only a core set of queries are selected and run on the clone. This can be insufficient for automatic indexing, particularly if DML queries are missed. Indexes can cause overhead on DML activities, leading to global workload regression even though some queries might benefit from the indexes. For these reasons, Oracle chose to implement automatic indexing on the production system directly. This gives the automatic indexing task full access to the real time data and workload, enabling it to make the most comprehensive decision (including but not limited to accounting for DML costs) and minimize index creation delay.

1.2 Challenges and Limitations

The major challenge of having everything run in the production database is to ensure minimal disruption to user workloads during automatic indexing actions. To avoid performance regressions, the index candidate should only be visible to the validation component but not visible to user queries until verified. No component of the automatic indexing task should compete with the user workload for CPU or IO resources. Additionally, automatic indexing should not disrupt user workload either, e.g. holding up locks resulting in user query waits or causing large user query compilation spikes.

Many applications rely on expressions in their queries, like UPPER(), SUBSTR(), DECODE() etc. Not having support for expression/function-based indexes would be a key limitation of any automatic indexing feature.

1.3 Oracle's Approach

In production systems, we have implemented an always-active background automatic indexing task that frequently monitors the application workload, creates indexes that are deemed useful, validates the impact of the indexes on the workload, and decides whether to keep or revoke the decision to use the indexes based on performance and resource metrics.

Oracle automatic indexing supports indexes that potentially improve performance of queries containing equality and range predicates, function-based predicates including JSON predicates, and a combination of predicates involving multiple columns. The optimizer's capability to track expressions and column groups in the workload allows automatic indexing to create and suggest indexes on expressions and group of columns. Automatic indexing components, like validation of candidate indexes or dropping of unused indexes, are agnostic to index types – empowering the task to be extensible and able to easily incorporate new index types in the future.

The automatic indexing task is not limited to creating indexes based on the application workload, but includes all activities related to indexes, such as disabling (make UNUSABLE), enabling (REBUILD), hiding (make INVISIBLE to optimizer), and dropping. For example, the index should be automatically dropped if it is unused past a certain amount of time to clear its maintenance overhead. It requires no user input for its activities. It also provides various display options to view a report of its actions. All these index DDLs could affect user workloads, as they are simultaneously occurring in the production database, so we make sure to create and verify indexes in a non-intrusive manner. The creation of new indexes or dropping of existing indexes can invalidate current optimizer decisions and execution plans. These decisions and plans are called shared cursors in Oracle's terminology. Shared cursors are managed based on the automatic indexing action that just took place (section 6.2). Automatic indexing could be a resource-intensive process, so the task is designed to run in the background by capping resources under Oracle Database Resource Manager [50]. Since the task runs in the background with resource control plan, there would be no/minimal overhead on the workload in terms of latency or throughput during any stage of the automatic indexing pipeline.

Our automatic indexing task is flexible and designed to be independent of workload type, i.e. working well for OLTP workloads and not imposing any unnecessary side effects – like index maintenance overhead during data loads involving bulk inserts – for OLAP workloads. The latter is achieved through accounting for DML maintenance cost during validation of a particular index candidate before and after rebuilding it (sections 5.3, 6.1.1).

Index validation/verification is done on a per-statement basis. Index advisors seem to perform better over query-level selection than workload-level selection [53]. Considering the cost and benefit of all indexes globally may help reduce the number of indexes, but in our experiments, we didn't see many more indexes than what would have been manually created for optimal performance (exemplified in section 7.2). Additionally, delineating the start and end times of a particular workload is not clear. Even with the workload time period established, if we wait for it to execute entirely and then select best indexes, we might miss a chance to benefit a top query seen in the beginning of the workload run. Currently, we deem an index as beneficial if it improves the statement by a threshold percentage. There could be cases where an index might not benefit a statement by enough margin, but still help improve the workload by a lower percentage and thus not be created. That is the trade-off we chose, as we saw it more advantageous to benefit TOP resource consuming queries faster in the workload. Furthermore, if a decision affects another statement negatively, Oracle's SPM (section 6.3) will help avoid any global regression to the workload that could be caused by automatic indexing.

2 AUTOMATIC INDEXING METHODOLOGY

All automatic indexes are non-unique b-tree indexes, which can be partitioned or non-partitioned. Indexes are non-unique to prevent any failure of user DML inserting rows with duplicate keys. Although we could create unique automatic indexes if the indexed

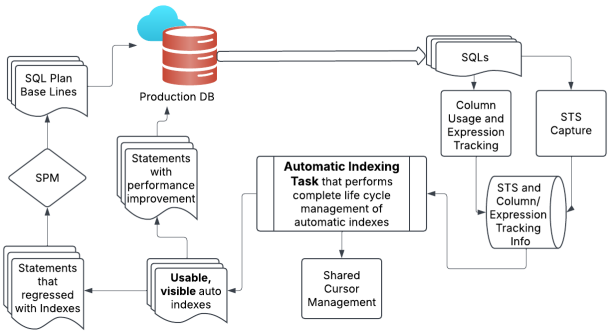


Figure 1: Automatic Indexing Components.

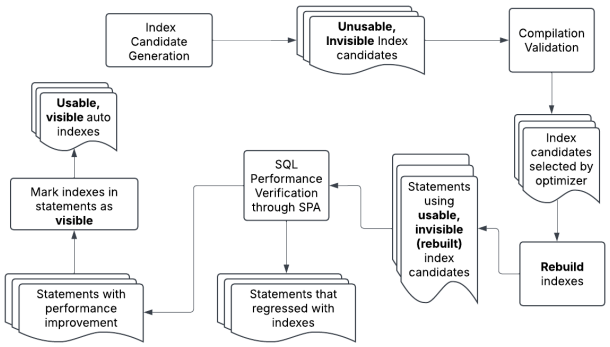


Figure 2: Automatic Indexing Task.

column is declared unique and non-null, the extra work to maintain/drop the uniqueness if the constraints on column are later dropped does not seem worthwhile. In this section we give an overview of our automatic indexing feature.

The high level idea is while the user workload is running, the automatic index task is invoked periodically (e.g., every 15 minutes) and runs with a time limit (e.g., 1 hour). In each iteration, the task creates a set of candidate indexes in UNUSABLE and INVISIBLE status based on column, column group and expression usage, which could be on single or multiple predicate seen in individual queries. Then it uses the top N past executed user statements to validate if the candidate indexes are worth rebuilding. The top N statements could be based on CPU, IO utilization and/or new statements seen after the previous iteration. Once the task reaches a steady state (analyzed most of the workload) most runs every next 15 minutes become a no-op except when the task re-validates some of the verified statements to see if anything has changed.

The validation of candidate indexes consists of two steps: (1) compilation (to check whether the optimizer would choose the candidate indexes); (2) execution (to check the performance of the SQL statements after rebuilding the candidate indexes). For each statement, after validation, there are 3 possible outcomes:

(1) The statement improves above a performance threshold: All the indexes rebuilt for that statement are marked as VISIBLE, and available to use in the workload.
(2) The statement regressed below a performance threshold: The regression is avoided in the workload using the SQL Plan Management (SPM) feature (section 6.3). SQL Plan baselines (section 6.3) are created for known verified good performing plans and sub-optimal plans with those automatic indexes are prevented. The performance of the statements is monitored in real time outside of the automatic indexing task and SQL Plan Baselines are created automatically.
(3) The statement performs within the performance thresholds: No action will be taken for this statement and the indexes will remain INVISIBLE.

Finally, the use of all automatic indexes is monitored, and an index will be dropped if it has not been used for a long time. Automatically created indexes may not be used if:

- Application SQLs are removed or changed.
- Data distribution changed.
- Automatically created SQL Plan Baselines prevent all plans using an automatic index.

Several automatic indexes may be merged into one, and some may even be dropped, if they have similar column sets or expressions and if a subset of those indexes is sufficient to give optimal performance.

Figure 1 highlights the key automatic indexing components, and Figure 2 shows the automatic indexing task in more detail. SQL Tuning Set (STS) and SQL Performance Analyzer (SPA) [49] mentioned in these figures are objects used to facilitate index validation, and are further defined in sections 5.1 and 6, respectively. The following four sections detail each of the main stages of our automatic indexing workflow, chronologically.

3 CANDIDATE GENERATION

We track columns, column groups and expressions used in predicates during optimization. The automatic indexing task generates index candidates based on tracked information in equality predicates, equality joins, range predicates, and predicates involving expressions (including JSON) on user created schemas. These candidate indexes are created in UNUSABLE and INVISIBLE mode. In this mode, the indexes will have only metadata. They will not consume any space and are not maintained during DMLs. They will not be used in user query executions. Statistics for these indexes are derived based on table and column statistics since the index is not materialized.

The search space of all index candidates could be quite large when considering all possible combinations of columns in all tables of a workload. The following subsections describe column usage and expression tracking in more detail, and showcase how they, along with other methods, work to constrain the candidate generation search space (also outlined in Figure 3).

3.1 Table Skipping

Indexes might not help or be usable for all tables. Oracle's automatic indexing does not create index candidates for temp/staging tables or non-internal tables (i.e. external tables, hybrid tables, etc.). Tables

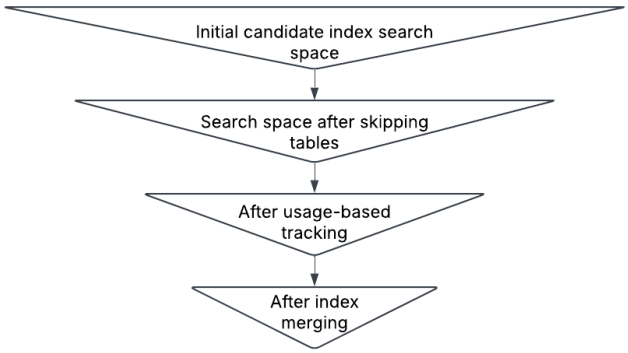


Figure 3: Candidate index search space reduction.

with no statistics or stale statistics (which are rare with Oracle's automatic statistics management) are also skipped, as statistics for meta-data only indexes depend on accurate table and column statistics – the lack of which can result in bad recommendations. Small tables (e.g. all blocks of the table can be fetched in a single IO) are excluded as well. In this case, indexes are not very useful as scanning the table incurs only a single IO and is better than fetching index blocks which can result in multiple IOs. Users can also specify a table exclusion list so automatic indexing does not consider those tables.

Candidates from any highly DML active tables are automatically skipped from creation as well. The impact an index can have on DML activities is further described in section 6.1. A table can be classified as highly DML active if at least one of two are true: 1) every week, every row in the table churns through DML changes at least once (the number of rows undergoing DML changes is greater than or equal to the current total number of rows in the table), or 2) the majority of the DMLs on the table are bulk loads (the number of rows undergoing inserts is greater than or equal to 50% of the current total number of rows in the table). Bulk loads usually involve large amounts of data and have a global impact. For example, an application might be written in a way that nightly data analysis can be only issued after bulk loads have been completed. Index maintenance overhead on bulk loads can be quite significant, affecting the application waiting for the load to complete. Hence it is best to avoid index maintenance during bulk loads.

3.2 Usage-Based Searchspace

As mentioned above, columns, column groups, and expressions are tracked for those only appearing in predicates. Furthermore, automatic indexing only looks at equality, equality joins, LIKE/range, and expression predicates. Non- equality predicates apart from range predicates are typically not used for accessing indexes, and hence not used by automatic indexing. They are not used by other areas using column group usage either, so are not tracked. The resulting index candidate set from this tracking is a small subset of all indexes possible, i.e. when looking at all combinations of columns in tables. For example, if there is table t1 in a workload, with columns c1, c2,...c10, but only two queries:

(1) Select count(*) from t1 where c1 = 5 and c4 = 'John';

(2) Select c4 from t1 where c1 = 2 and c5 > 10;

The index candidate pool can be limited to only sets including c1, c4, and c5. Indexes on the other unused columns will not be helpful. Column and expression tracking helps to identify index candidates that might benefit the workload and eliminates those that will surely not.

3.2.1 Column and Column Group. Column usage recording occurs during query optimization. We internally maintain two structures, one to hold all column usage and the other to hold all column group usage for a statement. We record this usage at the end of optimization – accounting for whether a column is part of a column group or not. Whenever a query is optimized, the columns used in predicates will be recorded in these internal structures. The columns or group of columns are identified when we examine predicates for selectivity estimation. Column usage is tracked from all types of predicates, while column group usage is tracked only from equality and range predicates. For example:

Q1: select prod_id, amount_sold from sales s, customers c where s.channel_id = 'S' and s.buyer_id = c.customer_id and s.seller_id = 5001 and s.amount_sold > 50 and s.prod_cost > 100;

Table 1 shows single columns usage tracking for 'Sales' table and Table 2 shows column groups' tracking for 'Sales.'

We can generate two multi-column index candidates on columns (channel_id, buyer_id, seller_id, amount_sold) and (channel_id, buyer_id, seller_id, prod_cost).

An equality predicate (join or non-join) column is considered part of a column group if there is at least one other equality predicate or range predicate. Indexes are created on a single column if it's not included in a group. For columns that are tracked as part of a group, index candidates will be generated with the concatenation of the columns while each individual column that is part of a column group will not have a corresponding candidate generated. Rational is, if you have an index on multiple columns, optimizer is not likely to pick the index on individual columns when the predicates involve multiple columns. If there is more than one range predicate but no equality predicate, the columns in the range predicates will not be tracked as part of a group. Instead, multiple individual candidates will be created on each of the range predicates based on just column usage (no group). Rationale is, range predicates on multiple columns cannot be used effectively for scanning leaf blocks satisfying the predicates. For the example query Q1 above, if a multi-column index on all equality and range columns (channel_id, seller_id, amount_sold, prod_cost) is created, predicates on channel_id, seller_id, and amount_sold can be used to build a key to access the relevant index rows (referred as access predicates) satisfying the predicates on channel_id, seller_id, and amount_sold. The first range predicate after equality can be used as an effective access predicate. The predicate on prod_cost (filter predicate) will be mostly evaluated after identifying the index rows satisfying predicates on channel_id, seller_id, and amount_sold. Once we have a range for one column, predicates on subsequent range columns will not be effective in pruning index leaf blocks. The goal is to create candidates with columns from access predicates that help reach index rows directly relevant to the query. Oracle b-tree index implementation also does not support specifying two range keys for index scans, but they can be specified for post index filters

Table 1: Tracking column usage based on predicates they appear in.

Table name	Column	EQ predicates	EQ-join predicates	Range predicate
Sales	channel_id	Yes	No	No
Sales	buyer_id	No	Yes	No
Sales	seller_id	Yes	No	No
Sales	amount_sold	No	No	Yes
Sales	prod_cost	No	No	Yes

Table 2: Column Group Tracking.

Table name	Equality Columns	Range Columns
Sales	channel, buyer, seller	amount_sold, prod_cost

(note that Oracle supports multiple range keys for bitmap index scans using AND-OR bitmap logic). Automatic indexing focuses on improving scans. For the above example, we create two multi-column candidate indexes on (channel_id, seller_id, amount_sold) and (channel_id, seller_id, prod_cost). All equality columns are present in each index as prefix followed by one of the columns of range. Currently we do not build candidate indexes on columns from group by, order by etc. Indexes on predicate columns that select small sets or rows from tables tend to have higher impact on OLTP query performance than indexes that avoid sorting needed for group by/order by operations. The cost of sorting is not significant if done on small row sets. We also currently do not consider "covering" indexes that have all columns referenced in the query and avoid table row fetches. These indexes are usually wide and have high storage and maintenance costs.

3.2.2 Expression Tracking. Oracle's automatic indexing also supports function based indexes to benefit queries having certain expressions on columns. Certain expressions like upper(), substr(), decode(), expressions needed for JSON etc. are heavily used by customers/organizations, hence our motivation to create index candidates on commonly used functions. For example:

Q2: select * from employees where empno = 1 and deptno = 2 and upper(ename) = 'JOHN' and salary+10 > 100;

A single-column index candidate on the entire expression 'upper(ename)' will be generated as well along with the multi- column index candidate (empno, deptno). The 'addition' operator is one of the expressions we do not automatically create a function-based index for, so 'salary+10' will not be an index candidate. Expressions are not part of column group tracking explained in section 3.2.1, so the multi-column index candidate [empno, deptno, and upper(ename)] will not be generated at first, but it may be created in later automatic indexing iterations through index merging – further described in section 3.3.

Support for function-based indexes is enabled through expression tracking – a generally useful feature whose clients are not restricted to automatic indexing. Expressions are very important constructs in SQL queries. They could involve simple operators such as "+", "*", or more complicated ones such as PL/SQL functions. They could appear anywhere in a query including where clause, select list, group by, having clause, etc. Popular expressions could appear in multiple queries, and in multiple parts of the same query. Oracle tracks several aspects of these expressions automatically – the text of the expression, its average evaluation cost, evaluation count etc. Tracked expressions come from one or more columns of the same table. They are identified at compilation time and statistics like average evaluation cost and count are captured at end of executing statement. Expressions that can be considered as automatic function-based index candidate (i.e. common functions that are used relatively recently) are flagged as such during tracking. Automatic indexing candidate generation algorithm will create index candidates for such flagged expressions.

3.3 Index Merging

If a user workload is large, the number of candidate indexes generated could also be large, which will increase maintenance cost. To reduce the number of indexes without losing functionality, we can merge two index candidates during candidate generation if the columns in one index are the prefix of the other (i.e. one covers the other). For example, an index created on column (b) will be merged by an index created on columns (a, b), assuming the order of a and b can be exchanged. The new index will be on (b, a), and order of the columns cannot be changed later. In this case, b has to be in leading position of the index to cover the cases where the index on b was useful. For indexes created on columns of equality predicates, the order of the columns in the index doesn't matter, but if there is a column in a range predicate, its position cannot be changed – it must be put in the end. Rationale is given in section 3.2.

We also drop an existing automatic index i1 if in a later iteration of automatic indexing, another index i2 is generated that covers i1. The covering index should also have a higher or equal status to the existing index, i.e. index status UNUSABLE is lower than VALID which is lower status than VISIBLE. This includes if i1 contains a function based index column as well. Function-based indexes are basically indexes on virtual columns created on expressions. The virtual column will be available after function-based index creation, and can be tracked like any other column, i.e., it can be tracked in column group usage as well. Continuing the section 3.2.2 example, the next time Q2 is optimized, the grouping (empno, deptno, virtual_column_upper_ename) will be recorded and the next automatic index iteration can create the merged index.

4 INDEX CREATION

We create candidate indexes as UNUSABLE and INVISIBLE indexes. Indexes created in UNUSABLE state have no storage structures hence no maintenance cost. Also, the INVISIBLE indexes are not

available to the user workload until verified. Indexes will be created under the same owner as table owner. By default, it will be created in the same tablespace as the table resides in. A preference can be set to create automatic indexes in a predefined tablespace.

Optimizer is invoked in a special mode to consider these UNUSABLE and INVISIBLE indexes as part of compilation validation (described in section 5.2). The more accurate index statistics are, the better index access path decisions the optimizer can make. The statistics for UNUSABLE indexes are derived since the index itself is not materialized yet. The number of index keys (NDK) is same as number of distinct values (NDV) of the column of single column index. For multi column index, we compute approximate NDV [51] of column groups by scanning the table and use it as NDK of the index. If there are n candidate multi column indexes to create for a statement, we just need to scan the table once. Other index statistics (leaf_blocks, clustering_factor, etc.) are derived based on NDK.

For partitioned tables, local index will be created if table partitioning column list is a prefix of the index column list; otherwise, a global non partitioned index will be created. In a local index, all keys in a particular index partition refer only to rows stored in a single underlying table partition.

If optimizer picks the candidate indexes and generate plans based on estimates, automatic indexing will rebuild (materialize) these indexes. The index segments are created and will inherit all other storage properties (compression, etc.) of the tablespace it belongs to. Accurate statistics of these indexes are computed as part of rebuild. These accurate statistics will then be used for SQL performance verification (section 6), remedying any slight inaccuracies that might have resulted from using derived statistics during compilation validation. In one of our workload environments, for example, performance verification was done for 62,723 statements. It produced the same plan for 54,504 of these statements as in compilation validation using derived statistics, and a different plan for 8,219 statements. This shows that derived statistics were accurate about 87% of the time, generating the same plan as when optimizer had real index statistics in performance verification – establishing a high correlation between the derived statistics and actual statistics.

There is a specified space budget that can be consumed by automatic indexes. Before rebuilding the index, we will check the current total size of all automatic indexes and the budget. We estimate the size of the index and if its addition exceeds the budget, the index is not rebuilt. If some other indexes are dropped later (e.g. when an index is not used for extended period), we may have more space and then we can create and rebuild new indexes. All this information is logged for diagnosing/informational purposes.

Index rebuild can take considerable amount of time. Automatic indexing will perform several such index DDL operations. Oracle has the capability to do these operations “online” and allow other DML operations on the table while these index DDL operations are performed. Automatic indexing performs all DDL operations using the "online” option, avoiding impact to the customer workload.

4.1 Index Pruning

Some workloads have tables with large number of columns and many statements with multiple filter combinations of these columns along with various equijoin conditions. Automatic indexing can create a large number of indexes on these combinations of columns if they benefit the relevant statements. With the creation of several automatic indexes, the compilation time will also increase since the index search space is larger. The optimizer will have to cost all applicable indexes before choosing the best one. For example, if a statement s1 has a predicate on a column c1 that is a leading column index i1(c1, c2, c3), then index i1 may be used for s1 and will be costed. With a large number of indexes, several such indexes could be applicable to a single query, and a lot of time could be spent on optimizer index costing.

Automatic index merging (section 3.3) can reduce the number of automatic indexes and can reduce the overhead of costing all these indexes to a certain extent. However, there can still be a large number of automatic indexes after merging. Hence, we also prune automatic indexes to be considered by optimizer access path selection based on their relevance during compilation of SQL statements. We use a heuristic-based approach to determine the best automatic indexes in certain categories. Only the best in each of the categories will be considered for access path selection. One such category is all statements involving equality predicates. The heuristic is as follows: For each column referenced in the predicates in the category, if an index contains leading columns matching the referenced columns, then derive the number of distinct values of the conjunction of those columns. The index with the highest number of distinct values is considered best (most selective index), and costed. The other indexes with matching columns are not costed. The best indexes in other categories are found similarly. We have 5 such categories, so the heuristics try to cost at most 5 indexes. Evaluation section 7.6 shows the positive effect of automatic index pruning on a customer workload. These heuristics are used for automatically created indexes, but can potentially be used for manually created indexes in the future.

5 INDEX VALIDATION

For the set of candidate indexes created, we want to know which of them will be used by the optimizer and if the performance of the statements improves after creating the indexes. To validate the indexes, we capture past executed statements from user workloads in a SQL Tuning Set (STS) [1], described in section 5.1. To find out whether the optimizer will use the candidate indexes, we use SQL Plan Analyzer (SPA) [49] to compile the statements in STS and check if the indexes are used in the plans, as detailed in section 5.2. We only rebuild the indexes that are used in the plans. REBUILD changes index status from UNUSABLE (disable) to VALID (enable) but keeps index mode INVISIBLE.

5.1 Capture SQL Workload

SQL tuning set (STS) [1] is a database object that includes one or more SQL statements, along with their plans, execution statistics and execution context. Oracle has an Auto STS created out of the box and used by many tuning features, including automatic indexing. SQL statements can be loaded into an STS from different sources; for Auto STS, it periodically captures the user’s workload from shared cursor cache. For any statement already existing in the STS, all the performance metrics in STS will be replaced with the new values from shared cursor cache. Any statement in STS not executed

within a certain retention period will be purged. Statements that are similar in nature are also purged more frequently. We use 373 days as the retention period to cover one full year with one week overlap. This value allows to capture any statements that are run in the past year. With this retention period, statements and plans that run on a particular day of the year (for example queries run from end of the year financial report) are not dropped from STS. Automatic indexing will not drop indexes built for these statements as it detects that the indexes are still used by the plans.

5.2 Compilation Validation

The automatics indexing task (and its underlying stages such as validation) is a repeated task. In each iteration, we are only interested in the new statements which have not been validated yet. We also want to put emphasis on expensive/often repeated statements as opposed to statements that may only get executed once (ad-hoc). Therefore, we select the statements in the STS that have not been selected before, rank them based on some metrics (a function of elapsed time, number of executions, etc.) and choose the top N to be compiled. As the task is iterative, all statements are bound to be evaluated at some point. The compilation is done using SQL Performance Analyzer (SPA) [49] in a special mode where UNUSABLE/INVISIBLE indexes are considered by optimizer. The goal is to check if these indexes will be picked by optimizer. The candidate indexes with UNUSABLE status used in the compiled plans are rebuilt in INVISIBLE mode. Statistics of non- partitioned indexes, or (sub)partitions of partitioned indexes, are gathered as part of rebuild. If we rebuild an index that covers an existing index, we drop the existing index in the next iteration given the existing index has a lower or equal status as the new index to reduce the total number of indexes created and save space. The rebuild may not succeed if doing so exceeds the index space budget allocated. This usually might happen for statements in later iterations, but by that time indexes for top N statements would have already been rebuilt and improved overall system performance.

Initial candidate indexes that are UNUSABLE (meta data only) and INVISIBLE can be considered as the list of all the potential indexes needed for the workload. Automatic indexing uses the optimizer to pick the best indexes from this list in a cost-based manner that can be rebuilt and marked as VISIBLE for the workload. Optimizer uses pruning techniques to reduce the execution plan search space if it is large due to complex nature of the query with large number of joins, large number of indexes etc. Using the optimizer guarantees that we pick relevant indexes that would eventually be used to execute the SQL in an efficient manner. Internal workings of the Oracle optimizer is beyond the scope of this paper.

5.3 Maintenance Cost-based Pruning

Automatic indexing accounts for index DML maintenance cost when verifying indexes (detailed in section 6.1.1). Before that, in compilation validation itself, it also prunes the indexes used in compiled plans whose estimated maintenance cost is more than maximum possible benefit. This is to avoid automatic index rebuild altogether and save resources if maintenance cost is too high.

We use similar IO costing and execution statistics as in the verification costing (section 6.1.1) to avoid rebuilding indexes under

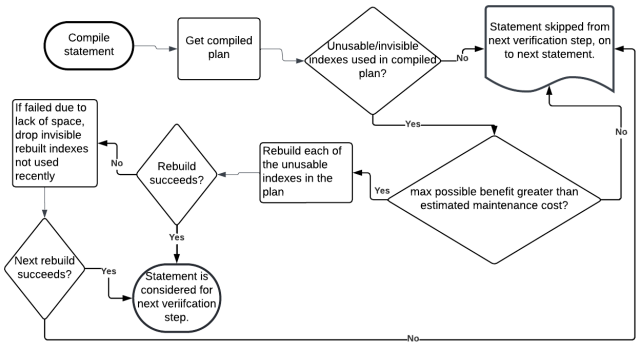


Figure 4: Compilation validation workflow.

certain conditions. Oracle has infrastructure to track number of DMLs (insert, update, delete) for each table. With approximate index levels and number of DMLs per day, we can compute the buffer gets/IO needed to maintain the index in a day. We have the performance numbers like statement buffer gets per execution without the candidate index in STS for prior executions of the statement. We can compute average daily buffer gets for this statement from STS. By having the new candidate index, the total buffer gets per day can be brought down to 0 in the best case, i.e. the maximum benefit possible is the average daily buffer gets for the statement. If the cost (in terms of buffer gets/IO) of maintaining the new candidate indexes per day is higher than the average daily buffer gets, rebuilding index will never be cost effective. We avoid rebuilding such candidate indexes. We only consider buffer gets here as CPU cost computation with auto indexes (section 6.1.1) needs index rebuild cost per row, which is not available before rebuilding the indexes. This pruning occurs right after compilation validation, but before rebuilding candidate indexes, as captured in the high level workflow in Figure 4.

For evaluation, in one of our workloads before pruning, we noticed 24 automatic indexes created and rebuilt, but not marked VISIBLE. These indexes would be marked UNUSABLE after 1 day if they didn't benefit any other statements later in the workload, but during that 1 day period where they were VALID, they seemed to cause regression. After pruning, these 24 automatic indexes are now not even rebuilt, avoiding any maintenance overhead regression.

6 SQL PERFORMANCE VERIFICATION

Compilation validation checks that the optimizer will choose the candidate automatic indexes. But the performance impact of that choice is unknown. By executing the statements using SQL Performance Analyzer (SPA) [49], we can discern this impact.

SPA enables assessment of system changes' (e.g. new indexes) impact on the response time of SQL statements. Once the indexes are validated and rebuilt after compilation validation, SPA will execute all the statements that have gone through compilation validation and picked some UNUSABLE indexes. This SPA execution is done in a special mode with all rebuilt automatic indexes (not yet VISIBLE to customer workload) made available to optimizer. This "execution verification" checks if the new plan with auto indexes is performing better (or worse) with respect to cpu time, buffer gets,

etc., than the original plan in STS. Based on the outcome of the performance comparison, the indexes become VISIBLE (if performance is better) or remain INVISIBLE (if performance is similar or worse) as detailed in section 2.

When the statement is executed in SPA, it is run with a time limit computed as average run time of the statement in the past plus a threshold. If execution exceeds the run-time limit, it is an obvious regression so no need to run the statement to completion and waste resources. This is all done in the background by the automatic indexing task.

6.1 Index Maintenance

Indexes can cause overhead on DML activities. This can lead to global workload regression even though some queries might benefit from the indexes. Our automatic indexing recommends indexes considering net benefits, i.e., accounting for the cost of maintaining the created indexes as part of DML along with their benefits. This means that when a statement is deemed to be improving with automatic indexes after execution verification by SPA [49], the overall benefit (cpu time, buffer gets improvement) of the indexes outweighs overall DML maintenance cost of the indexes.

6.1.1 Cost-Benefit Analysis. Execution verification provides a mechanism to measure statement query performance gains by automatic indexes. It compares the average performance metric (cpu_time and buffer_gets) of the plans of previous executions from STS with that of the execution metric obtained by running the statement with automatic indexes using SPA. The difference in these metrics multiplied by number of executions per day provide average daily benefit (in terms of cpu_time and buffer_gets) for the statement. The buffer gets benefit can be easily converted to IO benefit.

The challenge is how to measure index maintenance cost. Running DML statements in the background execution verification process, unlike running SELECT queries, can interfere with foreground operations. The alternative of trying out DML queries with and without the indexes on a clone (an approach used by others as also mentioned in section 9) would be too expensive, let alone inconsistent with our current automatic indexing which works without cloning. We opt for a more lightweight approach with a proxy for measuring maintenance cost (CPU and IO) rather than directly executing the DML. Using optimizer cost can also be ruled out, as optimizer cost for DML statements do not necessarily reflect the actual cost of maintaining the index. In most cases, the index maintenance is the last operation in the plan and this operation cannot be avoided for DMLs and computing cost is not necessary as there is no other alternative.

To compute daily CPU cost, we get the maintenance cost per row and multiply it by the number of daily DML changes. Automatic index candidates are rebuilt if they are used in compilation validation (section 5.2). We can get the total CPU time from the execution statistics of INDEX REBUILD operation and compute per row build cost. Each automatic index's daily CPU maintenance cost can be calculated as [the total number of DMLs on a table prorated over a 24-hour window (average daily DML count)] times a DML factor times the per row build cost of index rebuild. The DML factor is used to map per row REBUILD cost to per row DML cost. We then sum the CPU maintenance costs of all the automatic indexes used in the statement to get total daily CPU cost.

The total daily index IO cost of an index can be calculated as the average daily DML count for the table times [the index b- tree level minus an index level caching factor]. The caching factor indicates the number of index levels cached and thus not contributing to index IO cost. We again sum the IO cost of all automatic indexes used in the statement to get the total daily IO cost.

For each statement, we will mark all its automatic indexes VISIBLE to the workload only if their total daily CPU and IO benefit is greater than the total CPU and IO cost, respectively, in a 24-hour time window. We noticed performance regression in some DML statements after enabling auto indexing (AI), without accounting for DML maintenance cost. For example, CPU time for one of the DMLs regressed to 154,972 ms compared to baseline 75,524 ms. Buffer gets also regressed from 7,755K baseline to 15K. With accounting for DML maintenance costing, we do not see these regressions anymore in the workload.

6.2 Shared Cursor Management

Oracle stores the optimizer decisions and plans into a structure called a cursor. All cursors are stored in a shared memory area of the database server called the Cursor Cache. The goal of caching cursors in the cursor cache is to avoid compiling the same SQL statement every time it is executed, and instead use the cached cursor for subsequent executions of the same statement. However, cached cursors may not be used, and compilation could happen for many different reasons. Statistics gathering on database objects, existence of new available indexes on a table, different NLS setting parameters, and bind/host variables are some factors that may force creation of a new cursor. Based on the validity of old cursors, the compilation can be classified into two categories:

(1) Correctness driven compilation - If no valid old cursor is available in the system, an immediate compilation must take place. Invalid old cursor is either non executable or may lead to wrong results. For example, if an index is dropped, the old cursor using that index is no longer executable.
(2) Performance driven compilation - In some situations, old cursors are still valid, but may lead to subpar performance. For example, a newly created index results in a better performing plan. It is still valid to use the existing cursor without the new index, and not necessary to immediately build a new cursor.

The following two sections describe techniques to avoid compilation and building a new cursor.

6.2.1 Deferred Cursor Invalidation. Automatic indexing creates new indexes in UNUSABLE and INVISIBLE mode. It does not need to create a new cursor for this operation as this index is not VISIBLE and cannot be used in the workload. Once the indexes are marked VISIBLE after index verification, the new indexes can improve query performance. Compiling and generating new cursors are required to see this improvement. However, if the table for which a new index is available is used in several statements, and if the system starts compiling all those statements at once, there will be a spike in compilation load and system performance can degrade. For cursors that need performance driven compilation,

we use a deferred invalidation method. The database assigns each cursor that needs to be compiled a randomly generated time period. SQLs affected at the same time due to the new index typically have different time periods. Compilation occurs only if a query accessing the cursor executes after the period has expired. In this way, the database diffuses the overhead of compilation over time.

6.2.2 Non-Blocking Compilation. Deferred invalidation evenly distributes the compilation of several cursors of different statements. However, when multiple sessions concurrently execute a specific statement and if it needs to be compiled, only one session will compile and build a new cursor; all the other sessions must wait until the new cursor is ready. In highly concurrent systems, this wait event would be significant. Most of the compilation needed for automatic indexing is performance driven. In this case, executing an old cursor without using the new index is still valid. Non-blocking compilation technique is used to avoid the wait event. The idea is to further extend the life-time of the cursors that require performance driven compilation. When multiple sessions concurrently execute the same statement, one session is appointed to perform the compilation and new cursor creation. Before starting the compilation, the selected session extends the life span of the old cursor, making it still shareable by other concurrent sessions. Other sessions will share the old cursor and will not be blocked by the compilation and new cursor creation in the selected session.

Non-blocking compilation and deferred cursor invalidation (section 6.2.1) are used by other areas that require performance-driven compilation as well, like statistics gathering, where we do not need to cursor invalidate and build a new plan.

6.3 SQL Plan Management

SQL Plan Management (SPM) [44, 45] provides a framework to preserve current SQL execution plans amidst system and data changes while allowing new plans that are verified to have better performance. It allows for a controlled evolution and use of better performance plans for a SQL statement. With SPM [44, 45], the optimizer automatically manages execution plans and ensures that only known or verified plans are used. These plans are captured as SQL Plan Baselines [44, 45]. When a new plan is found for a statement, it will not be used until it has been verified to perform better than the Baselines created.

Automatic indexing creates an index and marks it as VISIBLE after it verifies that the index improves the performance of the statements. However, if performance verification finds that the statement has regressed, it will create a SQL Plan Baseline corresponding to plans that performed better without the newly created automatic indexes. This SQL plan baseline could contain other automatic indexes verified to improve performance in previous automatic indexing iterations. Optimizer will only pick plans from available SQL Plan baselines and avoid the suboptimal plan with new indexes. Even if these indexes go on to be made VISIBLE for some other statement later, this statement will not use them, and instead will execute according to the created SQL plan baselines.

Statement performance is monitored in real time outside of the automatic indexing task, and SQL Plan Baselines are created as necessary to prevent regressions for statements not verified by automatic indexing task yet. Plan Baselines may also be created if other changes in the system (e.g., data distribution changes) render an auto index to be unbeneficial and causing regression now.

7 AUTOMATIC INDEXING IN THE FIELD

Automatic indexing feature has been in production for more than 6 years [46]. The following aspects of automatic indexing are evaluated in publicly available and Oracle customer workloads: automatic indexing performance benefit and identification of unused indexes in a well-tuned application (NetSuite), automatic indexing with high DML workload and significance of function-based indexes in untuned application (Swingbench), automatic indexing in a hybrid (Data warehouse + OLTP) workload, resource consumption of automatic indexing (section 7.4 and 7.5), and effect of automatic index pruning on compilation time (section 7.6). In sections 7.1-3, automatic indexing was evaluated as follows:

(1) Run workload and capture performance numbers. This workload run will use manually created indexes (if any). This run is to get baseline performance numbers without any automatically created indexes.
(2) Enable Automatic Indexing.
(3) Rerun the workload after implementing recommendations from automatic indexing, which includes creation of new indexes and deletion of indexes that are not used by workload. Get the performance numbers.
(4) Compare performance numbers from 1 and 3.

The experiments were done to evaluate the impact of automatic indexing on Oracle customer workloads specifically. The goal is to automatically help improve Oracle customer workload performance through indexes. There are customers currently deploying auto indexing in production by default or via phased rollout and continue to use it. Their identities and evaluation metrics are not disclosed for confidentiality reasons.

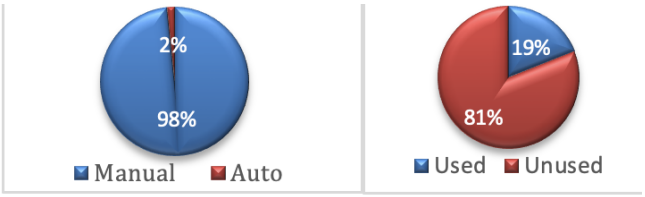
7.1 NetSuite

NetSuite [47] is a unified business management suite, encompassing ERP/Financials, CRM and ecommerce used by more than 37,000 customers. Complex applications like NetSuite are already tuned with indexes. These indexes are manually created during development and maintenance of the application over a long period of time. As mentioned in section 1, there are cases where manual creation is insufficient. Indexes being missing usually causes performance issues, and having experts manually diagnose and add them takes considerable amount of time and effort. Another issue is that while new indexes are created over time for different cases, some of the old available indexes may not be useful anymore. These unused indexes use some space and will unnecessarily incur maintenance overhead during DMLs.

Automatic indexing was evaluated using a typical NetSuite database workload to see if it:

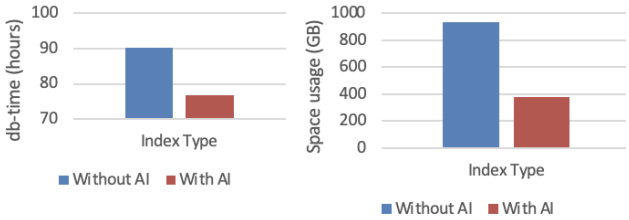
- creates additional indexes needed for the workload.
- detects indexes that are not used anymore.

Figures 5a and 5b show some metrics captured from this NetSuite evaluation. There were about 3600 manually created indexes in the system. Automatic indexing was creating a small number, 259 (7%) of additional indexes and marked 59 (2%) of them VISIBLE. The



(a) System indexes. **(b) Manual index usage.**

Figure 5: NetSuite Evaluation Index Details.



(c) Performance and index space usage with/without automatic indexing

Figure 6: NetSuite Evaluation Results.

indexes that are INVISIBLE will be dropped automatically. So just about a 2% increase in the number of indexes in the system as pictured in Figure 5a helped improve performance of the workload by 15%, depicted in Figure 6c.

Automatic indexing detected that 81% of those manually created indexes were not used in the workload and recommended to drop them (Figure 5b). Users can set a retention period preference for automatic indexing to drop all indexes unused past the retention period. This 81% of indexes occupied 60% of the entire index space. This 60% of index space can be reclaimed by dropping the unused indexes. These indexes are created at application development stage to cater to the indexing needs of large set of NetSuite customers who have different data distributions. Some of these indexes may not be useful for all customers. Also, these indexes are created over a long period of time as the application is evolved over years. For example, the developers may create an index on a set of columns initially and later create another index that has a superset of columns or completely different set of columns due to application statement changes. Automatic indexing can detect new indexes that cover multiple manually created indexes. It is difficult to detect and drop indexes that are not helpful due to new indexes or indexes not needed for the specific customer and evaluate the impact manually. Automatic indexing helps achieve this automatically, and this space savings' potential is highlighted in Figure 6c.

7.2 SwingBench

Swingbench [48] is an OLTP benchmark designed to stress test Oracle database. Some small customers may not have resources to tune the application and create necessary indexes. The Swingbench we used was approximately 3.3GB in size, having 11 total tables, 33 SQL statements, and 27 manually created indexes. Out of the 27, 15 were constraint enforcing indexes and the other 12 were secondary indexes. To simulate the "untuned" application environment, we

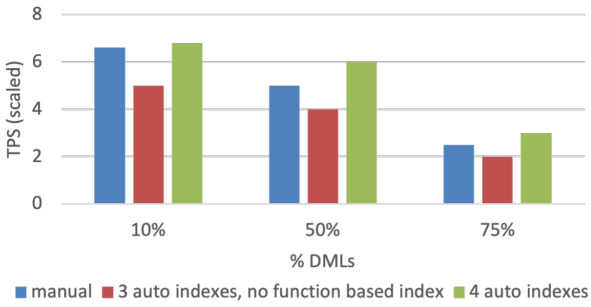


Figure 7: Manual vs. automatic index Transactions Per Sec.

dropped all those 12 secondary indexes and evaluated automatic indexing. We also varied percentage of DMLs in the workload. This evaluation used Oracle Enterprise Linux machines, with Intel Xeon Processor with 24 CPUs. The experiment was using 50 concurrent user sessions. In this experiment, automatic indexing created 4 indexes.

Automatic indexing is able to provide better performance with a small number of indexes. Figure 7 gives the Transactions per Second (TPS) with manual vs. automatic indexes. The performance improvement with automatic indexing is higher as the percentage of DMLs increase in the workload. This is expected since more indexes incur maintenance cost with DMLs. In this specific experiment, we found that automatic indexing is creating the same set of 4 indexes for all the 3 DML percentages. That is, the benefit of these indexes in queries is higher than the cost of maintenance of these indexes during DMLs. One of the 4 indexes was a function based index. Some of the queries benefitted significantly from that function based index. We collected TPS after manually removing that one function-based index for 10% and 75% DMLs. TPS dropped by 24% for former and 40% for latter. Thus, support for function- based automatic indexes is very important for these kinds of workloads.

7.3 Hybrid Workload

We validated automatic indexing in a database that has a data warehouse and an OLTP workload (hybrid workload). The purpose of including the data warehouse workload was to measure the effectiveness of index maintenance overhead estimation on DML statements. These workloads have the following characteristics shown in Table 3. The evaluation was done on Oracle Linux Exadata X9M 8-node machine, with Intel(R) Xeon(R) Platinum processor.

For the OLTP workload, automatic indexing created additional 5 indexes on top of 234 indexes. Out of 5 new indexes, 2 of them got rebuilt, out of which 1 got marked VISIBLE. i.e. 2 of them used in compilation validation and only one of them passed execution verification. After the auto index is created, buffer gets improved by about 98% (from 18 to 0.37) and CPU time improved by 15% (from 5.95 to 5.1) in the workload. Figure 8 shows the number of indexes used before and after automatic indexing created the indexes, and their corresponding performance numbers (normalized metric unit).

Table 3: Hybrid workload configuration.

DB	Size (GB)	Tables	Manual Indexes	SQLs	DML
DW (Star schema)	81	47	0	130	10% of the fact table data is loaded in the workload
OLTP	18	355	234	4889	75% of the statements are DMLs

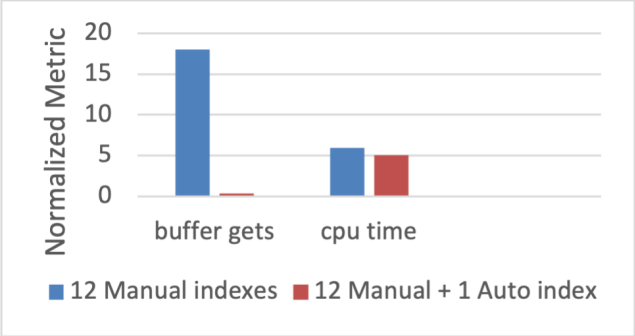


Figure 8: Manual vs. automatic index performance.

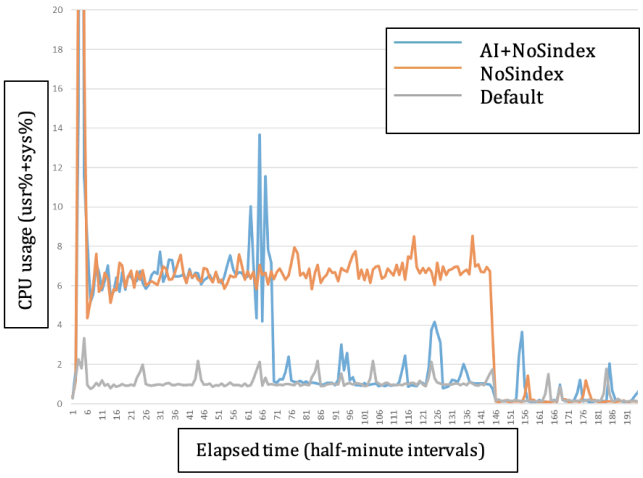


Figure 9: CPU usage percentage for 2-hour workload run.

For Data Warehouse (DW) workload, it created 180 UNUSABLE/ INVISIBLE indexes. However, these indexes did not get rebuilt since the cost of maintaining the indexes during loads came out higher than the maximum benefit for the statements with the indexes (5.3). Hence the performance of this workload saw no significant change before and after automatic indexing. Indexes would benefit if the DW was 'READ-only'. This experiment demonstrates that automatic indexing works well for hybrid workloads, creating necessary indexes for OLTP and avoiding them for DW with high loads.

7.4 Oracle Internal Workload

We validated automatic indexing in an internal email app workload (Beehive) to verify the percentage of CPU used by all processes (background and foreground) for the following configurations:

(1) Default – With all manually created indexes for application.
(2) NOSindex - Dropping all secondary indexes (manually created indexes except indexes enforcing constraints).
(3) AI+NoSIndex - Dropping all secondary indexes and running automatic indexing in the background.

The workload consumed 235 GB of memory, and was run on one node of Oracle Exadata X2-2 for 2 hours. The graph in Figure 9 shows the percentage of cpu (every ½ minute interval) used in the above configurations. The last 25 minutes of the graph is cropped but shows the same tapering of all three configurations.

CPU usage was high for the first 10 minutes (ramp-up period) for all configurations. Default configuration was better in CPU utilization during this time compared to other 2 configurations. Automatic indexing was able to create, build and validate the indexes needed by 70th interval (in 35 minutes). The CPU usage is higher for some intervals during this time for automatic indexing plus No secondary index configuration due to the CPU usage by the automatic indexing background process that rebuilds and verifies the statement. After 35 minutes though, the CPU usage of this configuration is similar to that of the default case. However, the no secondary configuration uses more CPU. This indicates that by this time, automatic indexing created all necessary indexes for the workload, producing better and efficient performing plans.

After minute 73 (ramp-down time), the workload was not using many indexes, so CPU usage was similar for all configurations. In this application, automatic indexing achieved similar performance as manually created indexes in a short amount of time by consuming just little more resources early on.

7.5 Automatic Indexing Task Impact

We evaluated automatic indexing's impact on a workload using the same SwingBench environment as in section 7.2. We used TPS as a measure of throughput to see if there is overhead in any stage of the AI life cycle. The results are captured in Table 4.

We varied 4 different setups to guage any TPS difference. Setup 1 was a baseline run. Setup 2 has an AI task running - AI task/DML overhead was expected since it would create/rebuild indexes amid the workload run. Setup 3 was again without any AI task, but with the indexes created in setup 2. The indexes would be unused but still valid so their maintenance overhead would be incurred by DMLs. Setup 3b is with the AI task running again. This was to see the overhead on DMLs plus any overhead from the task, which should be negligible as most of the necessary indexes would have been created in setup 2. Lastly, setup 4 was using the indexes created in setup 2 to show the overall benefit of AI. As expected, TPS decreases slightly from setup 1 to 2. The creation of new automatic indexes in setup 2 seemed to have some effect on DMLs, resulting in lower TPS in setup 3. 3b's similarity to 3 indicates that there is no additional impact from automatic indexing.

Table 4: Worklaod TPS.

No.	Setup Description	Average TPS
1	No AI Task, No indexes	45.67
2	AI Task, No indexes	44.2
3	No AI Task, setup 2 AI unused	43.79
3b	AI Task, setup 2 AI unused	44.31
4	No AI Task, setup 2 AI used	689

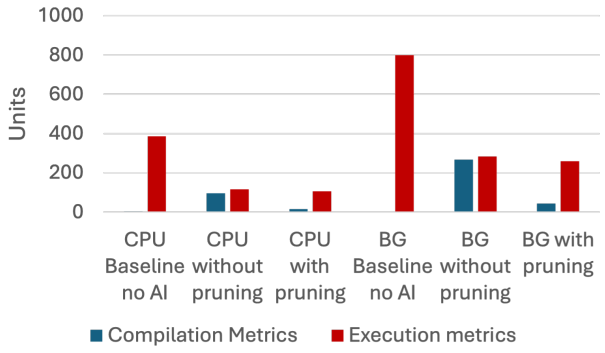


Figure 10: Pruning Effect on CPU time (min) and Buffer Gets

7.6 Automatic Index Pruning Effect

The effect of automatic index pruning (section 4.1) was evaluated on a scaled down version of a customer workload by measuring the total CPU time and Buffer Gets (with automatic indexes) in the workload with and without the heuristics. The workload had total size of 25 GB, with 25 tables, 2117 statements, and one table with 70 columns. Automatic indexing created 68 indexes, with each index on average containing 3.66 columns. This experiment was performed on Oracle Enterprise Linux machine with AMD EPYC Processor. As Figure 10 shows, automatic index pruning helped reduce compilation time by a factor of six (96 to 15 minutes) without introducing any regressions - total CPU time and buffer gets (scaled) also decreased. Baseline compilation numbers were too low to appear in the graph.

8 RELATED WORK

Automatically selecting indexes for a set of relational database queries (where the workload pattern is known), is a well- studied NP complete problem [17, 18, 19, 20, 21]. Most common index selection methods, especially in industry, solve the problem by selecting the set of indexes that would result in best performance given an existing, known workload. Contemporary index tuning techniques, including commercial tools [6, 7, 8, 9, 10, 15, 16, 23, 41, 55], follow this paradigm by relying on "what-if" analysis [43] to recommend the indexes with the most estimated improvement. "What-if" calls are one-step validation calls, either in execute or "no-execute" mode, to the query optimizer to get cost estimates for a particular index configuration. Existing state-of-the-art physical tuning tools mainly focus on assisting DBAs. Oracle's work builds on past research and existing tools to provide [one of] the first on-prem industrial strength, self-learning expert system for fully automated indexing. We have a two-step verification process that only uses "what-if" execution calls for configurations known to be picked by the optimizer, thus reducing the number of execution calls needed to make the best decision. Our validation also considers index maintenance cost during DMLs, a key limitation of existing advisors as they cannot handle workloads with Insert-Update-Deletes well [53].

Microsoft Azure SQL Database also has a fully automated indexing offering [3]. This offering and some others [52] uses a database copy which behind the scenes, receives customers' workload and provides auto-indexing service, differing from Oracle's in-production service (section 1). In Azure, if an index is detected to regress performance, it will immediately be dropped [3]. Oracle uses SQL Plan Management (6.3) to selectively use automatically created indexes in statements only if it improves the statements. Moreover, to the best of our knowledge, SQL Server does not create function- based indexes needed for workload automatically like Oracle. It has capability of creating computed columns [54] though, on which customers can create manual indexes.

Automatic index selection, and its variants [11, 12, 13], is still being researched [4, 5, 26], now with machine learning incorporated. Learned-based index advisors [14, 25, 40, 42] build machine learning models based on training workloads and use these models to select indexes and/or estimate index benefits [53]. Most of these algorithms still use the "what-if" optimizer-based index benefit estimation for index tuning. Although machine learning alternates to "what-if" calls/index benefit estimators have been proposed [24, 40, 42], these classification tasks just decide which plan, among two, has the cheaper execution cost and are mainly designed for query regression prevention. Moreover, most machine learning methods need to materialize all the index configurations, meaning training these models would be expensive and time-consuming. Learning convergence is also not guaranteed, with prediction quality only improving over time and initial predictions being mostly arbitrary and of low quality.

9 CONCLUSION

Oracle automatic indexing [2] is a fully automatic process that continually optimizes a SQL workload, offering net verified performance improvement. All stages (except workload capture, SQL Plan Management, Cursor Management) are performed by the automatic indexing task that runs periodically in the background with controlled use of resources to not impact normal user workload. All tuning activities can be monitored via activity reports, requiring minimal human interaction. It is also self-maintained, purging automatic indexes and any associated tasks, objects, and logs that have been unused/older than a specified retention period.

ACKNOWLEDGMENTS

We would like to thank members of the Oracle Optimizer, Indexing, Partitioning, Auto task, SQL Tuning Set, SQL Performance Analyzer, Functional Testing and Stress Testing teams for their contributions to various aspects of the product life cycle. We also thank Fusion Apps, SAP and NetSuite teams for helping test and validate automatic indexing.

REFERENCES

[1] Nigel Bayliss. 2020. What is the Automatic SQL Tuning Set? Oracle. Retrieved Jan 10, 2024 from https://blogs.oracle.com/optimizer/post/what-is-the-automatic-sql-tuning-set

[2] Arup Nanda. 2021. Automatic indexing with Oracle Database. Oracle. Retrieved Jan 10, 2024 from https://www.oracle.com/news/connect/oracle-database-automatic-indexing.html

[3] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu and Surajit Chaudhuri. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 666–679. https: //doi.org/10.1145/3299869.3314035

[4] Sheldon J. Finkelstein, Mario Schkolnick, and Paolo Tiberio. 1988. Physical Database Design for Relational Databases. *ACM Trans. Database Syst*. 13, 1 (1988), 91-128. https://doi.org/10.1145/42201.42205

[5] Michael Stonebraker. 1974. The choice of partial inversions and combined indices. *International Journal of Parallel Programming* 3, 2 (1974), 167-188. https://doi.org/10.1007/BF00976642

[6] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 146-155. http://dl.acm.org/citation.cfm?id=645923.673646

[7] Benoît Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zait, and Mohamed Ziauddin. 2004. Automatic SQL Tuning in Oracle 10g. In *VLDB*. 1098-1109. http://www.vldb.org/conf/2004/IND4P2.PDF

[8] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. 2000. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE*. 101–110. https://doi.org/10.1109/ICDE.2000.839397

[9] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. 2004. Database Tuning Advisor for Microsoft SQL Server 2005. In *VLDB*. 1110–1121.

[10] Adam Dziedzic, Jingjing Wang, Sudipto Das, Bolin Ding, Vivek R. Narasayya, and Manoj Syamala. 2018. Columnstore and B+ tree - Are Hybrid Physical Designs Important? In *SIGMOD*. 177-190. https://doi.org/10.1145/3183713.3190660

[11] Nicolas Bruno and Surajit Chaudhuri. 2007. An Online Approach to Physical Design Tuning. In *ICDE*. 826-835. https://doi.org/10.1109/ICDE.2007.367928

[12] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. 2006. COLT: continuous on-line tuning. In *SIGMOD*. 793-795. https://doi.org/10.1145/1142473.1142592

[13] Karl Schnaitter and Neoklis Polyzotis. 2012. Semi-Automatic Index Tuning: Keeping DBAs in the Loop. *PVLDB* 5, 5 (2012), 478-489. https://doi.org/10.14778/2140436.2140444

[14] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *CIDR*. https://www.cidrdb.org/cidr2017/papers/p42-pavlo-cidr17.pdf

[15] N. Bruno. 2011. Automated Physical Database Design and Tuning. *CRC Press, Inc., Boca Raton, FL, USA, 1st edition.*

[16] Pavle Subotic, Herbert Jordan, Lijun Chang, Alan Fekete, and Bernhard Scholz. 2018. Automatic Index Selection for Large-Scale Datalog Computation. *PVLDB* 12, 2 (2018), 143. https://www.vldb.org/pvldb/vol12/p141-subotic.pdf

[17] D. Comer. 1978. The difficulty of optimum index selection. *ACM Trans. Database Syst.*, 3(4):440–445.

[18] M. Ip, L. Saxton, and V. Raghavan. 1983. On the selection of an optimal set of indexes. *IEEE Trans. on Software Engineering*, SE-9(2):135–143.

[19] M. Schkolnick. 1975. The optimal selection of secondary indices for files. *Information Systems*, 1(4):141 – 146.

[20] J. Kratica, I. Ljubic, and D. Tos˘ic. 2003. A genetic algorithm for the index selection problem. In *Proc. of EvoWorkshops*, pages 280–290, Berlin, Heidelberg, 2003. Springer-Verlag.

[21] G. Piatetsky-Shapiro. 1983. The Optimal Selection of Secondary Indices is NP-complete. *SIGMOD Rec.*, 13(2):72–75, Jan. 1983.

[22] Pedro Holanda, Mark Raasveldt, Stefan Manegold, and Hannes Muhleisen. 2019. Progressive Indexes: Indexing for Interactive Data Analysis. *PVLDB* 12, 13, 2366-2377. https://doi.org/10.14778/3358701.3358705

[23] Sanjay Agrawal, Surajit Choudhuri, and Vivek Narasayya. 2000. Automated Selection of Materialized Views and Indexes for SQL Databases. In *Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt,* 496-505. https://www.vldb.org/conf/2000/P496.pdf

[24] Jiachen Shi, Gao Cong, and Xiao-Li Li. 2022. Learned Index Benefits: Machine Learning Based Index Performance Estimation. *PVLDB*, 15(13): 3950-3962. https://dl.acm.org/doi/10.14778/3565838.3565848

[25] A. Sharma, F. M. Schuhknecht, and J. Dittrich. 2018. The Case for Automatic Database Administration using Deep Reinforcement Learning. *arXiv preprint* arXiv:1801.05643

[26] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. 1997. Index Selection for OLAP. In *Data Engineering, 1997. Proceedings. 13th International Conference on*, pages 208–219. IEEE.

[27] S. Idreos, M. L. Kersten, S. Manegold, et al. 2007. Database Cracking. In *CIDR,* volume 3, pages 1–8.

[28] G. Graefe and H. Kuno. 2010. Self-selecting, self-tuning, incrementally optimized indexes. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 371–381. ACM.

[29] F. M. Schuhknecht, A. Jindal, and J. Dittrich. 2013. The Uncracked Pieces in Database Cracking. *PVLDB*, 7(2):97–108. https://people.csail.mit.edu/alekh/papers/p97-schuhknecht.pdf

[30] F. M. Schuhknecht, J. Dittrich, and L. Linden. 2018. Adaptive adaptive indexing. *ICDE*.

[31] F. Halim, S. Idreos, P. Karras, and R. H. Yap. 2012. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *PVLDB,* 5(6):502–513. https://dl.acm.org/doi/10.14778/2168651.2168652

[32] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. 2011. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 4(9):586–597. https://www.vldb.org/pvldb/vol4/p586-idreos.pdf

[33] S. Idreos, M. L. Kersten, and S. Manegold. 2009. Self-organizing Tuple Reconstruction in Column-stores. *SIGMOD*, pages 297–308.

[34] E. Petraki, S. Idreos, and S. Manegold. 2015. Holistic Indexing in Main-memory Column-stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1153–1166. ACM.

[35] H. Pirk, E. Petraki, S. Idreos, S. Manegold, and M. Kersten. 2014. Database Cracking: Fancy Scan, not Poor Man's Sort! In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, page 4. ACM.

[36] P. Holanda and E. C. de Almeida. 2017. SPST-Index: A Self-Pruning Splay Tree Index for Caching Database Cracking. In *EDBT*, pages 458–461.

[37] S. Idreos, M. L. Kersten, and S. Manegold. 2007. Updating a Cracked Database. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data,* SIGMOD '07, pages 413–424, New York, NY, USA.

[38] I. Haffner, F. M. Schuhknecht, and J. Dittrich. 2018. An Analysis and Comparison of Database Cracking Kernels. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*, DAMON '18, pages 10:1–10:10, New York, NY, USA. ACM.

[39] E. Teixeira, P. Amora, and J. C. Machado. 2018. Metisidx-from adaptive to predictive data indexing. In *EDBT*, pages 485–488.

[40] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *Proceedings International Conference on Management of Data, SIGMOD Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 1241–1258.

[41] Surajit Chaudhuri and Vivek Narasayya. 2020. Anytime Algorithm of DatabaseTuning Advisor for Microsoft SQL Server. (June 2020). https://www.microsoft.com/en-us/research/publication/anytime-algorithm-of-database-tuning-advisor-for-microsoft-sql-server/

[42] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow*. 9, 3 (2015), 204–215.

[43] Surajit Chaudhuri and Vivek Narasayya. 1998. AutoAdmin "What-if" Index Analysis Utility. *ACM SIGMOD Record*, 27(2):367–378. https://cs.brown.edu/courses/cs227/archives/2008/Papers/AutoAdmin/autoadmin_conf_version.pdf

[44] Nigel Bayliss. 2019. SQL Plan Management in Oracle Database 19c. *Oracle White Paper.* https://www.oracle.com/technetwork/database/bi-datawarehousing/twp-sql-plan-mgmt-19c-5324207.pdf

[45] Oracle. 2024. *Overview of SQL Plan Management.* Oracle. Retrieved March 13, 2024 from https://docs.oracle.com/en/database/oracle/oracle-database/23/tgsql/overview-of-sql-plan-management.html#GUID-F1C45056-F998-43E5-B362-83F88DA49E58

[46] Oracle. 2024. *Autonomous Database.* Oracle. Retrieved March 13, 2024 from https://www.oracle.com/autonomous-database/

[47] NetSuite. 2024. *The #1 Cloud ERP.* Oracle. Retrieved March 15, 2024 from https://www.netsuite.com/

[48] Swingbench. 2024. *Swingbench...* Swingbench. Retrieved March 13, 2024 from https://www.dominicgiles.com/swingbench.html

[49] Oracle. 2024. *Introduction to SQL Performance Analyzer.* Oracle. Retrieved March 14, 2024 from https://docs.oracle.com/en/database/oracle/oracle-database/19/ratug/introduction-to-sql-performance-analyzer.html#GUID-860FC707-B281-4D81-8B43-1E3857194A72

[50] Oracle. 2024. *Managing Resources with Oracle Database Resource Manager.* Oracle. Retrieved March 18, 2024 from https://docs.oracle.com/en/database/oracle/oracle-database/23/admin/managing-resources-with-oracle-database-resource-manager.html#GUID-2BEF5482-CF97-4A85-BD90-9195E41E74EF

[51] Hong Su, Mohamed Zait, Vladimir Barriere, Joseph Torres, and Andre Menck. 2016. Approximate Aggregates in Oracle 12c. In *Proceedings of the 25th ACM International Conference on Information and Knowledge Management*, pages 1603-1612. https://doi.org/10.1145/2983323.2983353

[52] Ritwik Yadav, Satyanarayan R. Valluri and Mohamed. Zait. 2023. AIM: A practical approach to automated index management for SQL databases. In *IEEE 39th International Conference on Data Engineering (ICDE)*, pp. 3349-3362.
doi: 10.1109/ICDE55515.2023.00257

[53] Wei Zhou, Chen Lin, Xuanhe Zhou, and Guoliang Li. 2024. Breaking It Down: An In-depth Study of Index Advisors. *PVLDB,* 17(10): 2405 - 2418 doi:10.14778/3675034.3675035

[54] Microsoft. 2022. *Indexes on Computed Columns.* Microsoft. Retrieved November 20, 2024 from https://learn.microsoft.com/en-us/sql/relational-databases/indexes/indexes-on-computed-columns?view=sql-server-ver16

[55] Sam Idicula and Haoyu Huang. 2023. *Overview of the AlloyDB Index Advisor feature and how to use it.* Google. Retrieved November 22, 2024 from https://cloud.google.com/blog/products/databases/how-the-alloydb-index-advisor-helps-make-smart-indexes