VLDB (1996) 5: 229–237

The VLDB Journal
© Springer-Verlag 1996

Query processing and optimization in Oracle Rdb

Gennady Antoshenkov[⋆], **Mohamed Ziauddin**

Oracle Corporation, New England Development Center, 110 Spit Brook Road, Nashua, NH 03062, USA; e-mail: zmohamed@us.oracle.com

Edited by C. Mohan / Received August 1994 / Acceped August 1995

Abstract. This paper contains an overview of the technology used in the query processing and optimization component of Oracle Rdb, a relational database management system originally developed by Digital Equipment Corporation and now under development by Oracle Corporation. Oracle Rdb is a production system that supports the most demanding database applications, runs on multiple platforms and in a variety of environments.

Key words: Relational database – Optimizer – Dynamic optimization – Sampling – Query transformation

1 Introduction

Oracle Rdb is a relational database management system running on multiple platforms that include OpenVMS VAX, OpenVMS Alpha, and Digital UNIX, with more platform coverage to come. Since all platforms share a common code base, the query processing and optimization technology is largely platform-independent, and further we will concentrate on the features common to all the platforms. On VAX and Alpha clusters, Oracle Rdb runs with applications using embedded SQL, SQL module language, dynamic SQL, interactive SQL, and any of the layered products such as Oracle RALLY. Distributed and heterogeneous query processing is done via DEC DB Integrator (also known as Access Works) that distributes the appropriate SQL query portions to be executed at multiple sites running different database products including Oracle Rdb, DB2, Oracle 7, and Sybase servers. DEC DB Integrator has its own optimizer, the description of which is beyond the scope of this paper.

The Oracle Rdb query engine is built to handle the high end applications, those that use large databases and require faster processing of simple and complex queries. With the customer database sizes approaching tens and hundreds of Gigabytes, we have experienced a sharp increase in demand for performance and robustness. In response, we have revised our query optimizer and regression test system and have made the following VLDB-specific improvements.

1. To compensate for unavoidable [Ant93a] compile-time optimization mistakes, we developed and implemented a new run-time (dynamic) optimizer that removed the major shortcomings of single table access. Dynamic optimization of joins is currently in the design stage.

2. For the cases of wrong join order and strategy selection we introduced a tool called Query Outline which enabled a manual execution plan specification.

3. We made it our design style to propagate each new optimization technique into all areas where it can be potentially used. Imagine a production database with a changing data distribution, with new tables added, and indexes redefined. In this scenario, with time, different system areas get emphasized and a particular optimization technique may easily fall out of reach and stop working. For example, given a sorted index on column X, a few ORed equalities on column X can be filtered by a single index scan using a range list technique described in Sect. 2. When the user replaces this sorted index with a hashed index, he expects a performance improvement for the equality list restrictions. However, if the sorted order on X is needed, and the range list technology is not propagated to hashed indexes, an unnecessary extra sort will be performed, causing a substantial performance reduction. We found that such performance disruptions, being unpredictable, are the most painful to our customers and must be avoided early in the development process, during design stage.

4. To achieve equally good performance for the low/high extremes of query complexity and data flow sizes, we decided to use different techniques for different query complexities or size levels. And, along with the multi-level techniques, we supplied a number of mechanisms for complexity escalation and reduction (shortcuts). For example, when building a table row filter out of the list of row IDs, we first collect (and sort) IDs in a tiny pre-allocated buffer. Upon overflow, we escalate the task and allocate a mid-size buffer, and upon the second overflow, we escalate further and build a hashed bitmap.

[⋆] Deceased: Oct. 1995

Correspondence to: M. Ziauddin

5. We improved our support of binary large objects (BLOBs) by introducing the write-once read-many (WORM) option and thus making BLOBs well suited for processing multimedia objects. Further, a tight integration of BLOBs into the relational engine was made possible by providing a choice of placing them into any storage area and referencing them from any table position. Based on this functionality, merge of voice, video, graphics, and text can now be easily achieved through a SQL Multimedia front end and retrieved and manipulated using both SQL and multimedia-specific facilities.

6. With each new product release, we started to run a number of large-scale benchmarks, including TPC-D, Wisconsin, Set Query, AS3AP [Gray93], and an internal complex query benchmark. These benchmark runs have quickly become an important source for the performance evaluation, but we still receive most of our detailed performance feedback from our customers. Because of that, we are still looking for more exhaustive performance testing tools.

The Oracle Rdb query processing component is designed to provide high execution speed for the full spectrum of queries, from the simple ones running in OLTP environment to the very complex decision support queries. Currently, queries with up to 128 joins[1] can be evaluated with the boolean restriction complexity limited only by the operating system stack bound and the overall query source size.

The query processor consists of two major parts, the query compiler and query executor, each of which has its own optimizer. Initially, the compiler was designed after System R [DaNg82] with a cost-based static optimizer. Subsequently, many new features were added and older ones were replaced in an evolutionary fashion. For instance, instead of storing the compiled query for further load and execution, Oracle Rdb stores a full or partial execution plan that also can be manually modified, providing (1) a selective manual optimization, (2) a faster user-assisted query compilation, and (3) less dependency on changes in the physical database design. The query executor, on the other hand, not only evolved with the improvement of old techniques and the addition of new strategies, but was also re-worked to accommodate a new dynamic optimizer that allows switching the strategies in the middle of query evaluation.

This paper contains an overview of the technology used in the query compiler and executor. The rest of the paper is organized as follows. Section 2 describes various query expression transformations useful for building an efficient execution plan. Section 3 outlines the facilities used in the cost-based execution plan selection by the static optimizer. Join and single table access strategies are presented in Sects. 4 and 5, respectively. The basic concepts of the dynamic optimizer are given in Sect. 6 below. Section 7 introduces a number of advanced features: BLOB support, constraints, triggers, stored procedures, and user-defined functions. Section 8 concludes the paper.

[1] To achieve an acceptable or fast query compilation time, large queries should be expressed using nested subqueries like views, or derived tables (SELECT ...)AS$t(f_1, \ldots, f_n)$. When using the derived tables, the total number of base tables and subqueries should not exceed 128. For example, 12 subqueries with the number of base tables in each subquery not exceeding 10 and with the total of 116 base tables, will compile in about half a minute, whereas compilation of a flat 116-way join will take forever. Views can be nested arbitrarily with the total number of base tables not exceeding 128. Compilation of a 128-way join composed of the 4-way join views takes less than a quarter of a minute

2 Query transformation

The earliest query transformation in Oracle Rdb takes place in the SQL front end. SQL text is converted into a compact direct polish notation, speeding up the message traffic to the query engine, especially for remote communications. During the initial compilation phase, the query parser converts "polish" queries into a linked structure of internal blocks well suited for fast query traverse and rearrangement. Also, at this phase, a number of semantically equivalent query transformations takes place based on query structural properties (and ignoring database properties like data distribution in tables and indexes), aiming at better query execution plan production. Other semantically equivalent query transformations are done during the execution plan selection, and further, during code generation.

To broaden the application of join optimization and to reduce the number of distinct, run-time operators, some SQL constructs like IN (SELECT ...) are transformed into joins. The other example is an enclosure (SELECT AVG(X) ...) delivering exactly one row into a boolean expression. Also, on many occasions, we find and remove the redundant items from query expressions. For instance, the query parser would detect a (useless) ORDER BY clause appearing along with a non-grouped aggregation and eliminate it from further processing.

The views and derived tables are represented as a group of join items and optimized as such, without unnesting. We use this approach deliberately for a number of reasons. First, it helps to cut down the combinatorial join search; second, it results in shallower query execution trees; and finally, it enables straightforward application of operators such as aggregation and LIMIT TO N ROWS to the joined group result. The drawback of this approach is that it restricts join order permutations, and therefore can result in potential loss of optimal join order. However, limited unnesting of subqueries is performed. A subquery with join dependency (i.e., correlated subquery) is moved upward as much as possible in the nested hierarchy while still maintaining the join dependency. The upward movement causes non-correlated subqueries to move up to the top level where they are stacked as leading items in an n-way join.

The query parser establishes project operations for each subquery, including only the attributes needed for expression evaluation (such as boolean) and for the final or intermediate result delivery. In the extreme cases, when no attributes are requested, e.g., SELECT COUNT(*) ..., Oracle Rdb finds the most compact index and scans it. If all columns in a project list are contained in an index, the query compiler marks the index for "index-only" access and the index scanner consequently avoids any row fetches. Following one of the most traditional query optimization principles, the compiler pushes selects down through subqueries, views, and unions, whenever possible. This kind of query transformation requires some non-trivial design when attribute names change across the subquery boundaries and also during the transition between union and join subquery types. After the join order is established, the final binding of the attributes becomes known, and the code generator pushes the newly exposed selections even further down into the table and index scans.

The other (classical) way of producing and utilizing extra selects is to derive them from the transitive closures of equality chains. In Oracle Rdb, the transitive closures are exploited to their fullest extent. For instance, transitivity in (T1.X=T2.X AND T2.X=T3.X) increases the join search space by adding an extra T1.X=T3.X join equality. boolean (T1.X=T2.X AND T2.X>10) adds an extra T1.X>10 restriction for table T1. Selectivities of the redundant conjuncts introduced by transitivity closures are ignored in the query scope where they are redundant. In the first example, assuming the join order join(join(T1,T3),T2), a partial solution join(T1,T3) will incorporate selectivity of T1.X=T3.X, and then the full solution join(join(T1,T3),T2) will incorporate selectivity of T1.X=T2.X, but not selectivity of T2.X=T3.X. In the second example, we incorporate selectivity of T1.X>10 into one join leg and selectivity of T2.X¿10 into the other leg and then compensate for one of the two selectivities at the join(T1,T2) application point. Though the extra selectivity is ignored at and above the join node, it has its expected effect on the strategy selection within a join leg, especially if table T1 or T2 happens to be a derived table or view.

At the early parsing stages, NOT operators present in boolean expressions are eliminated by using De Morgan's boolean transformations. After that, each IN predicate with a set of literals is converted into ORed equalities, and then during code generation, into a list of index ranges to be scanned in the forward or reverse direction (see range list zig-zag skip discussion below). ORed equalities of IN predicate that are also ORed with comparisons, BETWEEN, and other range-producing predicates applied to the same attribute are converted into a single common range list.

Even more sophisticated boolean transformations become necessary for improving the index scan for a composite index key. When several attributes comprise an index key and there are enough ANDed equalities specified for a sequence of leading key attributes, a multi-attribute range list can be constructed. In such cases, Oracle Rdb produces the range lists from both disjunctive and conjunctive forms of a boolean restriction. For instance, given an index on key X‖Y, restrictions X=1 AND (Y IN (5,7) OR Y>10) and (X=1 AND Y=5) OR (X=1 AND Y=7) OR (X=1 AND Y>10) will produce the same range list ([1‖5, 1‖5], [1‖7, 1‖7], [1‖11, 1‖ ∞]) where ‖ stands for logical concatenation.

Historically, boolean transformation has been an area of constant and intensive improvement. Everyone on the development team acknowledges its utmost complexity and importance. Projection of a boolean into a concatenated attribute sequence of a composite index key is a difficult and elaborate task by itself. But this task becomes even harder when the attributes with known and unknown values interchange during the join order permutation, making index applicability to appear and disappear, partially or fully.

One way of dealing with the boolean transformation complexity is to delay doing the full transformation until the variable binding time when all boolean variables except the index key attributes are assigned some values. The earliest variable binding can take place either during query execution, when a start of an index scan is initiated, or during the last stages of compilation, when the join order is already determined and all boolean operands are found to be key attributes or constants. We wrote prototype software for the binding time boolean transformation and found that: (1) all desirable value-based transformations took $O(n * \log(n))$ time, whereas similar variable-based transformations typically required a polynomial time, (2) deeper transformations are possible when dealing with values because values allow immediate evaluation and linear ordering, and (3) after the value-based transformation, an efficient index scan similar to the range list scan can be performed using the transformed boolean as a range list analog.

As an illustration of the importance of the "range list" transformation, consider a particular case of a combined NULL restriction X IS NULL AND Y IN (5,7). In Oracle Rdb, a unique NULL value is stored for each attribute type, which makes it possible to treat IS NULL as an equality. (Note that in accordance with SQL definition, a NULL value compares high with all non-NULL values.) This allows the compiler to transform the above boolean into a range list ([NULL‖5, NULL‖5], [NULL‖7, NULL‖7]) and make index scans very efficient. A frequent case of a concatenated attribute range 1‖5≤ X‖Y≤3‖7 has a lengthy SQL expression (X=1 AND Y≥5) OR (X>1 AND X<3) OR (X=3 AND Y≤7). Oracle Rdb creates a range list out of it, and during execution, performs a seamless single index scan over the original concatenated range. This method is also applied to and yields a fast index scan for a string concatenated SQL expression like S1‖S2 BETWEEN "aa"‖"bbb" AND "zz"‖"yyy". And further, LIKE predicates, whose patterns start with literal characters, are ANDed with a range or range list (for upper/lower case letters) derived from this leading literal portion in order to enrich the pool of range restrictions.

3 Execution plan selection

To determine a join order, Oracle Rdb performs an exhaustive search and produces a left-deep tree of two-way joins for each individual subquery. Each partial solution based on some join subset might or might not deliver an intermediate result in some order. To reduce the exhaustive search space, the query optimizer employs a number of different heuristics. For example, it uses a depth-first search to create a total initial solution for a subquery. All subsequent solutions, total or partial, are pruned if they cost more than a known total solution. Only the least-cost solutions and solutions with different "interesting" orders are considered for further extension, where an order is interesting if it can be used for enforcement of ORDER BY, GROUP BY, DISTINCT, or for a sort merge strategy execution.

The least expensive join orders are determined individually for each subquery: first for the innermost subqueries, then for the containing subqueries, until the main subquery is reached. Both intermediate data orders and equality transitivities are propagated globally across the subqueries. For example, a merge strategy can be selected for the containing subquery that uses the order delivered by an index scan of the contained subquery.

In Oracle Rdb, execution plan selection is based on the cost model where the cost of an operator represents the estimated number of disk I/Os needed to perform the associated task. The cost functions take into account savings in I/Os due to buffering of data as well as physical clustering of data. The primary source of input to the cost model is the table and index cardinalities. These basic statistics on tables and indexes are dynamically maintained with a precision of $\log_2(n)$, where n is the actual table or index cardinality. When needed, the precise values can be enforced by running a maintenance utility.

Part of the cost model is calculation of the clustering effect done for clustered record placement explicitly specified in the schema, as well as, for implicit clustering of indexes having some leading key attributes in common with the clustering index. Accidental clustering is accounted for during query execution by the dynamic optimizer using a learning mechanism. Detection of clustering typically yields a manyfold performance improvement. Large-scale benchmarks and customer feedback made us aware of why accidental clustering occurs fairly often. One of the reasons for accidental clustering is that some related attributes like serial number and manufacture date follow virtually the same order (but this knowledge never becomes part of a database schema). Another reason why unaccounted accidental clustering occurs is because the initial table load was done in the order of one or more attributes. There is a whole class of applications that load large tables overnight or on weekends for subsequent read-only use. Any hidden clustering in such cases remains intact through the duration of the table's existence.

To obtain a good estimate of the index range scan cost is almost a hopeless task, given that range boundary expressions contain variables. Except for common sense, our only guidance in picking the cost formula was balancing between over- and under-estimation reported by our customers. As it stands today, the cost of an index scan over a range or range list involves its estimation as a function of the number of "leading" equalities involved in a multi-attribute key restriction. To estimate the cost of repetitive single-key retrievals, the cost of a B-tree descent takes into account the tree height and a correction factor for the buffering effect. Coupled with the cost-based execution plan selection is a manual control over the choice of join sequence, join and table retrieval strategies, and sets of indexes to use for each retrieval. This control is a part of the Query Outline feature and it is fully generic, that is, any partial join order or selective recommendations on the strategy or index choice can be given. The optimizer will enforce the applicable recommendations and apply the regular cost-based procedure to resolve any unspecified optimization dimensions. We found this flexibility useful for the quality and performance of manual optimization, as well as for the partial strategy freeze that, unlike the traditional stored query plans, lives longer with a changing schema and continues to benefit from future query optimization improvements. In addition, Query Outlines greatly simplify and speed up testing and problem solving when a particular strategy has to be reproduced with a short query on a miniaturized database version. (The problem here is that the reduced db/query versions used in debugging tend to deviate from the original cost calculation and deliver strategies not identical to those needed for debugging.)

With an extended syntax SELECT ... LIMIT TO n ROWS, the number of rows delivered by such a subquery is restricted to no more than n rows. Similar restrictions are imposed (1) by constructs like EXISTS predicate implemented as a subquery to deliver no more than one row, (2) by an explicit user intention OPTIMIZE FOR FAST FIRST to terminate a query after receiving the first few records, and (3) by propagating the "fast first" optimization request (with uncertain n) down the execution tree. The opposite intention of reading all result rows can be specified explicitly as OPTIMIZE FOR TOTAL TIME, can be set implicitly by operators like sort and aggregates, and can also be propagated down the execution tree. Note that the above data flow limiting intentions come in two distinct types: certain (LIMIT TO n) and uncertain (FAST FIRST). The industrial query optimizers today mostly support a *certain* limiting type, ignoring the cases when *uncertain* limits play a dominant role. One of these cases is when an interactive user gets the first answer screen and decides with about equal probability to terminate the query or to run it to completion. Another case amounts to the limit propagation down through the execution tree, where according to our experiments [Ant93a] any certainty quickly deteriorates into a zipfean [Zipf49] type of uncertainty. Unlike in traditional implementations, the Oracle Rdb optimizer deals with uncertain limits directly by marking the uncertain execution tree areas and running a parallel competition algorithm, described further in Sect. 6.

A case of processing not more than one row is recognized by the optimizer and is assumed to deliver a sorted order, thus avoiding a redundant sort. During execution, sorts are fully avoided upon detection of zero or one incoming row. This shortcut may seem insignificant, but in practice it might yield a substantial performance improvement. For example, if a data flow is sorted within a nested loop and almost each inner loop iteration delivers one or no rows, the effect of the initial sort cost avoidance becomes quite noticeable. The optimizer also avoids redundant sorts when one data order subsets another (e.g., a sort for delivering an ORDER BY X,Y is not used if a Merge strategy within this subquery already delivers the order X,Y,Z). In addition, it combines the project order (from GROUP BY and DISTINCT) with the output order (from ORDER BY) into a single order, whenever possible. Limiting execution activities to a single row fetch takes place in some instances of aggregation, e.g., MIN(X) and MAX(X) do a single descent into an index B-tree when an index on X is available. For multi-attribute indexes (say on X,Y,Z) a single-descent MIN(Z)/MAX(Z) retrieval can be achieved if restriction X=8 AND Y=5 is part of the query's WHERE clause. When all columns of interest are part of an index then "index-only" retrieval is used, which avoids fetching rows altogether. boolean selections based entirely on columns of an index are evaluated as part of an index scan, thus avoiding redundant row fetches.

Selection of a locking strategy in Oracle Rdb can be controlled by the LOCK FOR UPDATE option. With this option specified, any rows retrieved from the stored tables are immediately locked for update. Without this option, rows are first locked for read and only those that survive all query restrictions have their locks upgraded to “update” just before the update is actually performed. When all rows touched during retrieval are going to be updated, all rows are automatically “update” locked, thus avoiding the costly lock upgrades.

4 Join strategies

Oracle Rdb supports nested loop and merge join strategies. With the n-way nested loop join, the join columns of the outer streams become available to the inner streams for selection. Therefore, the inner streams generally use indexed access to locate only matching rows. The non-correlated subqueries or single-row value expressions are stacked as outer streams of an n-way nested loop join each delivering a single row, so that no actual looping occurs but rather a collection of needed data elements takes place.

The two-way merge join works on two sorted streams that can come from a table retrieval, from other joins, or from a subquery. Groups of rows with matching duplicate keys on both sides are processed by a mini-nested loop mechanism. When the merge order is delivered by an index scan, the merge key value from the opposite side is passed to the index side and is further passed to the index scanner to perform an efficient skip of all lower key values. This type of merge is called zig-zag skip, reflecting a pattern of the key skip B-tree traverse.

A skip key can be passed to the index through several subqueries, including the cases of intermediate aggregations grouped by one or more merge key attributes. Consider the query SELECT A.X, (SELECT AVG(B.X) FROM B WHERE B.X=A.X AND B.X IN (1,10,100)) FROM A; where indexes on X are defined for both tables A and B. This query will use the Merge strategy for joining tables A and B. The AVG aggregate will perform grouping on attribute X because of the “correlation” equality B.X=A.X. Merge keys will be passed from side A over the aggregate and into the index scan of B, thus enabling the B scan to perform zig-zag skip to the value greater than or equal to the value passed from side A.

Merge with zig-zag skip is the ultimate strategy to use in demanding applications that provide the necessary indexes for joins and complex selections. Compared with the other two join strategies widely used in the industry, zig-zag merge efficiently skips over the mismatched key stretches on both join sides, whereas the nested loop join skips (rather, jumps) over the keys on one side only and hash join performs a complete scan of both sides. Today we use the key skip mechanism for two-way merge joins and for range list index scans (see below). In the future, we plan to use n-way common key merge joins that offer an opportunity to skip over larger mismatched key ranges and use hash joins in cases of index absence [Gra94] in order to avoid sorts that are expensive and offer little chance for skipping. Left, right, and full outer joins are supported through the use of the same nested loop and merge join algorithms. The mismatched key rows on one or both sides are padded with nulls and delivered. When the outer join predicate is composed only of join key equalities, groups of rows with matching duplicate keys on both sides are processed by a mini-nested loop mechanism. In the case of more complex outer join predicates, Oracle Rdb uses a combination of mini-nested loop and bitmaps in order to separate, and process individually, the semi-joins and anti-semi-joins on one or both join sides within an equi-join-key group. When the non-equality portion of the outer join predicate is applied to the cartesian product of groups of rows on each side having the same join key value, there might be rows in each of these two groups that are fully rejected from the cartesian product (these rejected rows constitute anti-semi-join). While evaluating the cartesian product, the two bitmaps, one per side, keep track of those rows, which become part of the semi-join. At the end of cartesian product evaluation, semi- and anti-semi-joins are fully separated and the NULL-padded anti-semi-joins are delivered as part of this outer join.

5 Single-table access

Single-table access in Oracle Rdb employs either a sequential scan of a table, or scan of one or more indexes with or without associated row fetches, or direct access of a row by the row identifier (RID) supplied as part of a table restriction. Sequential table scan is accelerated by performing asynchronous prefetches of buffers containing data pages. To avoid a full scan of a big storage area when it contains a mixture of different tables or indexes, small tables stored within that area are accessed using the most compact index, if available. If there is an opportunity to organize the retrieval in different ways and there is a substantial uncertainty in choosing the optimal alternative, then a generic retrieval strategy called Leaf is used as described in Sect. 6. Otherwise, the traditional sequential or indexed retrieval strategies are used.

Index retrieval supports the forward and reverse scans of sorted (B-tree) indexes and the exact match lookup of hashed indexes. The data rows may or may not be fetched depending on whether the needed columns are all part of the index key. The start and stop points of an index scan are controlled by the low and high bound key values of a single range or by those of a sorted non-intersecting list of ranges. To skip the gaps between the ranges, the index scanner checks the current index page for inclusion of the low bound of the next range, and upon failure, it descends a B-tree to this low bound (see also a similar zig-zag algorithm in [CHHI91]). Restrictions on index key attributes not reducible to ranges are checked before a row is fetched. Note the positive performance impact of the ORed range list scan. The scanner opens the index only once, touches only the relevant pages, moves in a given direction without looping back, and delivers the records in a sorted order. In addition, Oracle Rdb exercises the same retrieval tactic for hashed indexes when a restriction is composed of ORed key equalities. For the query SELECT * FROM T WHERE X IN (7,2,5,0); with the hashed index on X present, the scanner will sort the key equalities into the equality list (0,2,5,7), open the index only once, retrieve rows for each key value, and deliver them in the sorted order. The equality list tactic applied to hashed indexes made some of our benchmark queries run several times faster due to the sort avoidance.

Application of the above scan tactic to both sorted and hashed indexes illustrates the advantage of the technology propagation approach mentioned in the Introduction. Consider for instance, a customer parametric query SELECT X,COUNT(*) FROM T WHERE X IN (:p1,:p2,:p3) GROUP BY X ORDER BY X; with a sorted index defined on X. This query does only one scan and aggregation and runs very fast. Then the customer decides to improve the performance by replacing the sorted index X with the hashed index X. Without the technology propagation, he would do three individual scans, one for each parameter, and then sort the unionized result. Performing sort, especially with many duplicates present, would reduce the query speed rather than increase it. With the range list technology propagated to hashed indexes, no extra sort would be needed, and a direct hash bucket fetch (1 I/O) would be performed instead of a B-tree descent (several I/Os), yielding the expected query speed increase.

When several ORed parts of a restriction can be resolved by scanning different indexes, the appropriate multiple index retrievals are performed and their result streams are concatenated. Possible multiple delivery of the same rows is avoided by applying a negation of restrictions on the previous streams to the current stream. Note that NULL value treatment in this arrangement should be the opposite to the way it is done in regular boolean evaluation. For obvious efficiency reasons, Oracle Rdb uses several types of data compressions. Table rows have their repeated characters compressed on the user's request. Indexes, in addition to the traditional prefix and tail compression [BaUn77], provide (1) order-preserving repeated character compression, (2) support for indexing of first few characters of long attributes, and (3) elimination of an attribute's guaranteed constant common prefix within the valid value range. All three non-traditional index key compressions were implemented upon customer request. For example, in case (1), an index of company or person names expressed in a Japanese (4-B Kanji) character set was considered the backbone of physical database design by our Japanese clients. The names needed a maximum of about 200 B to be reserved with an average of 8–12 B utilized for the name and the rest of the space filled with blanks. The composite index keys had a telephone number or some other items following the name attribute, thus excluding the possibility of trailing blank compression. Without blank compression, the index grew several times the table size for tables compressed by the repeated character encoding. After we discovered and implemented an order-preserving repeated character encoding, indexes shrank by a factor of ten and the entire database was reduced to a fraction of its original size.

6 Dynamic optimization

Traditionally, execution plan selection done during query compilation time uses a variant of the Selinger et al. [SACL79] cost model, propagating mean cost estimates through intermediate query results into the final query cost. However, the validity of this model has recently been seriously challenged [IoCh91], [Ant93a] by proving that the estimation error grows much faster than the rate sufficient for plan selection stability. In fact, assuming unknown correlations between intermediate data flows, the most common query operators like AND, OR, JOIN, UNION, EXISTS tend to quickly degenerate arbitrary-precise estimates into fully uncertain, zipfean[2] [Zipf49] probability distributions when the number of query operators grows.

The effect produced by, say, the AND operator on different probability distributions of selectivities of the two AND operands is not easily observable because manual calculation of it is difficult (next to impossible) and special software must be written to run the desired experiments. To see the blurring impact of ANDing, consider a boolean R1 AND R2 where restrictions R1, R2 can be enforced by scanning two different indexes and a precise selectivity estimate of 20% is available on each restriction. The selectivity probability distributions $p_{sel(R1})$ and $p_{sel(R2})$ have 100% probability concentrated at point 0.2 and have zero probability everywhere else in the selectivity interval [0,1]. A realistic assumption of overlapping of selected rows could be: rows fully overlap yielding the result selectivity of 0.2 (case of +1 correlation); rows overlap partially yielding 0.04 selectivity (0 correlation); rows do not overlap yielding 0 selectivity (-1 correlation); all other overlapping degrees are equally probable. Calling this an *unknown* correlation assumption, the calculated selectivity probability distribution of R1 AND R2 with unknown correlation between R1 and R2 happens to be a curve closely resembling a hyperbola, with a peak at zero selectivity. This fundamental property of the ANDed distribution is largely unknown in the research community, but the consequence of it is that even knowing selectivities on the two indexes one has no realistic guidance to judge whether two indexes should be scanned or only one. To make the situation worse, different combinations of different index portions may correlate differently, making precalculation of correlations on all index pairs a task of a great magnitude.

Another source of uncertainty has to do with a request for the fast first record delivery described in Sect. 3. We assume that in this case the number of rows requested from an execution tree node before a forceful node closing has a hyperbolic probability distribution. The rationale for this is simple: the probability of being asked for exactly ten rows is much higher than for exactly a million rows. In the index retrieval case, a request for ten rows should be satisfied best by scanning through the first ten index entries and immediately fetching and delivering data rows. On the contrary, a request for a million rows is best processed by scanning the index, collecting row IDs, sorting them, and then fetching data rows from the ordered sequence of pages, thus reading each page only once. So, once again, we will never know in advance which of these two strategies should be selected.

Given the impossibility of an adequate compilation time estimation even for moderately complex queries, we turned to dynamic optimization methods [GrWa89], [Ant91], [Roy91] that use the alternative plans for query and its parts, and that change the plans during execution. Today,

[2] The degenerated uncertain distribution tends to closely resemble hyperbola (a particular case of zipfean distribution) which is a concave function, concentrating half of the probability around zero and spreading the other half to the right of zero area. In contrast, "certain" distributions tend to be approximated by a bell shape which is convex and concentrates almost all probability in a limited area around the bell center.

Oracle Rdb employs dynamic optimization for single-table retrieval encapsulated in the Leaf retrieval strategy (Leaf strategy nodes always appear at the leaf level of the execution tree because they access base tables, not views or other types of subqueries). The Leaf retrieval design is based on the "competition" architecture and cost model described in [Ant91], [HoEn95], [Ant93a], [Ant93b].

The idea of competition is simple. Suppose that two different processes can accomplish the same task and the cost probability distribution of one or both processes is zipfean. Since for zipfean processes probabilities of being very inexpensive are significant, it makes sense to run them both simultaneously (or a single zipfean alone) for a small amount of time just sufficient to hit all those cases where the task can be accomplished quickly. If during this time no process completion is encountered, then we have to guess and continue only one process that we think has a lower cost. The trick here is that this second phase, being costly, is entered only on those occasions when the cheap and highly probable first phase fails. This advantage offsets the expense of a redundant first-phase process run, given enough skewness of the cost hyperbolic distribution. Competition also stabilizes the system behavior because the number of the mistakenly chosen long processes becomes reduced. Now, consider once again two different index retrieval strategies suitable for the "fast first" delivery request. Index scan with immediate row fetches has a hyperbolic cost probability distribution because cost is proportional to the number of requested rows, which we assume, has a hyperbolic distribution. Index scan that sorts all selected row IDs before row fetches and delivery has a very substantial initial cost of a complete index scan and row ID sort, so its cost distribution is not a zero-peak hyperbola. In the Leaf strategy, we use the following competition arrangement of these two basic strategies. First, we scan, fetch, and deliver data until the row count hits some threshold – this is a first competition phase exercising a single zipfean process. Then, if the threshold is reached, we switch to the bulk row ID accumulation, sort, and final row fetches. With this arrangement, we might lose on multiple data page reading during the first phase, but simultaneously, we avoid the large expense of the second phase, when in fact only a few rows are requested. It often happens that two or more indexes are available for conjunctive retrieval. In such cases, the Leaf strategy collects the row ID lists from the indexes and intersects them following the basic idea of Babb's arrays [Babb79]. What is unique to our intersection technique is that we scan simultaneously the two indexes estimated as most selective and collect row IDs into two in-memory buffers while there is sufficient space available. This way we often discover a shorter index quickly, even if our original estimates where incorrect, and then do further row ID list intersections faster because the consecutive intersections can become only smaller. Compared to the "fast first" competition, the conjunctive two-index competition runs two scan processes during its first phase, not one.

When resolving zipfean uncertainties, a more efficient competition can be arranged if, during the first phase, a running process can also deliver a reliable estimate of the cost of some other activities to be performed in the future. In the Leaf node, an index scan collects (and also intersects on the fly) a row ID list. With each new row ID, an almost precise estimate of the number of reads necessary for the final row fetches is calculated. This I/O estimate is compared against the cost of the currently best strategy for the entire table retrieval. If the projected final fetch cost closely approaches or exceeds the cost of the best available strategy, the dynamic optimizer abandons this index scan as unproductive and starts another scan if more indexes are available. Here, competition takes place not on the basis of the process completion, but between the two projected costs. Given that the index scan cost might often be a mere fraction of the final fetch cost (one index page might contain a hundred row IDs), the first phase extra scan overhead is reduced accordingly.

In addition to the three competition types described above, several other types of competition take place and interact in the Leaf strategy, covering major combinations of (1) whether row fetches via a given index are needed or not; (2) whether a particular order is expected to be delivered or not, and (3) whether the fast first row delivery is requested or not. To further improve the competition efficiency, a sequence of index involvement is revised at each retrieval start by quick probing into the index [Ant93a]. In the future, we plan to achieve more precise estimation via index sampling (the efficient sampling techniques and stop rules are available in [OlRo89], [Ant92], [LiNS90], [HaSw92]).

Improving estimation efficiency done at the table retrieval start will certainly transform some of the hyperbolic cost distributions into more certain bell-shape functions. This will eliminate the need for running some of the competitions. At the same time, the estimator itself will absorb many of the potentials for simultaneous exploration of the least expensive retrieval arrangements. One simply cannot afford to get engaged into sampling from one index when a single descent into a leaf B-tree node of the other index can uncover a single result row or emptiness of the result, especially when fetching singletons constitutes the bulk of a workload. We see competition being a major contributor to efficient estimation. And, apart from estimation, some hyperbolic uncertainties like those encountered in the presence of the "fast first" request will never go away.

As for the past, before the dynamic optimizer introduction in 1990, we performed the ANDed multi-index retrieval by scanning the indexes and sort-merging their RID lists. This strategy, being efficient when the optimal subset of useful indexes is selected correctly, caused us a lot of trouble because a reliable optimal subset selection was impossible to achieve at compile time. The mistakes of selecting the wrong index subset dropped the performance by a few decimal orders and literally forced us to shift the selection decision into the run time.

The dynamic optimization approach we have chosen is geared to deal with high uncertainties at execution time by using parallel runs and by using an entirely dynamic cost model. This contrasts the Leaf retrieval strategy with a table retrieval approach in [MHWC90] that also switches the strategies, but relies on the statically calculated thresholds for making the switching decisions. Compared to the multiple precalculated plan approach [GrWa89], [INSS92], we tend to assume that the dynamic removal of high uncertainties would open a much larger space for the execution plan rework than it is possible to exhaust by the limited number of precalculated alternatives. Also, dynamic plans are as vulnerable to estimation uncertainties as the static optimizer. Thus, we see it to be more advantageous if the full scale query and execution plan transformation is performed at execution time.

7 Advanced features

Following customer requests, a number of advanced features were designed into the product. From the very beginning, starting with its first release, Oracle Rdb supported BLOBs that can be stored as chains of manageable byte sequences called segmented strings. Segmented strings are used for storing text, multimedia and other objects of arbitrary size, and are updated on a per-segment basis. Special provisions are made to enable an efficient storage of BLOBs on the WORM devices [Ren92].

Oracle Rdb also supports integrity constraints and triggers. Several types of triggers are supported, such as AFTER INSERT, BEFORE UPDATE, AFTER UPDATE, and BEFORE DELETE. A trigger can nest other triggers as long as recursion does not take place. Special-purpose constraints such as primary key, foreign key, NOT NULL, and UNIQUE constraints are supported in addition to general-purpose CHECK constraints. Special-purpose constraints are optimized so that they are evaluated only when it is necessary, depending on the update action. For example, primary key, NOT NULL, and UNIQUE constraints need not be evaluated when a row is deleted from a table. Likewise, a foreign key constraint need not be evaluated when a row is deleted from a foreign key table. To further improve the constraint enforcement efficiency, the user has been given an option to evaluate the constraints either upon operating on a row or at the transaction commit time. It makes sense to use the latter option for a bulk integrity check when the expectation of a constraint violation is low. In the bulk check case, the constraint optimizer collects a RID list of rows to be verified, thus reducing the check set size from an entire table to a much smaller set of rows.

Two important features implemented in Oracle Rdb in response to customer requests are stored procedures and user-defined functions. The abstract data types are targeted for the near future. Stored procedures are part of the client-server architecture that aims at reducing the message traffic. User-defined functions and data types open the way for a much deeper integration of various applications into the database engine. Other advanced features include the support for distributed transaction update capability using a two-phase commit protocol, and the internationalization support using different collating sequences and character sets.

8 Conclusion

Today's requirements on database management systems, which are rooted in the information explosion phenomenon, can be condensed into a single statement: various entities should be pushed to their extremes. Among the extremes are (1) storing and processing very large data collections, (2) manipulating large objects of a diverse nature, (3) integrated support of a huge number of users, (4) high processing speed, and (5) support of growing query complexity.

To fulfill these requirements for query processing and optimization, we channel our major effort into a few main directions. We deliver support for the new extended functionality, following closely the evolving SQL3 standard. We also work on incorporation of different types of parallelism and, as a part of the performance increase effort, we migrate a number of boolean transformations from the compiler into the run time, with the net effect of a larger key skip at index scan and significant optimizer simplification. To efficiently store and process large tables and indexes, we made inroads into new areas of data compression. This includes (1) bitmap compression with fast set operations [Ant95] and (2) order-preserving "dictionary" compression [ALM94], which matches the existing order-indifferent "dictionary" compressions in speed and compression rate.

References

[ALM94] Antoshenkov G, Lomet D, Murray J (1994) Order-preserving key compression. Technical report. DEC Cambridge Research Laboratory, Cambridge, July

[Ant91] Antoshenkov G (1991) Dynamic optimization of a single table access. Technical report DBS-TR-5, DEC-TR-765. DEC data base systems group, Cambridge, July

[Ant92] Antoshenkov G (1992) Random sampling from pseudo-ranked B+ Trees. In: Proc 18th VLDB Conf, August, Vancouver, Canada

[Ant93a] Antoshenkov G (1993) Dynamic query optimization in Rdb/VMS, In: Proc 9th Int Conference on Data Engineering, April, Vienna, Austria

[Ant93b] Antoshenkov G (1993) Query processing in DEC Rdb: major issues and future challenges. IEEE Bull the Techn Comm Data Eng 16: 42–52

[Ant95] Antoshenkov G (1995) Byte-aligned bitmap compression (poster abstract). In: IEEE Data Compression Conf, March, Snowbird, Utah

[Babb79] Babb E (1979) Implementing a relational database by means of specialized hardware. ACM Trans Database Syst 4: 1–29

[BaUn77] Bayer R, Unterauer K (1977) Prefix B-trees. ACM Trans Database Syst 2: 11–26

[CHHI91] Cheng J, Haderle D, Hedges R, Iyer B, Messinger T, Mohan C, Wang Y (1991) An efficient hybrid join algorithm: a DB2 prototype. In: Proc 7th Int Conf Data Engineering, Kobe, April

[DaNg82] Daniels D, Ng P (1982) Query compilation in R*. IEEE Database Eng 5: 15–18

[Gra94] Graefe G (1994) Sort merge-join: an idea whose time has passed? In: Proc Int Conf Data Engineering, Houston, Texas

[Gray93] Gray G (ed) (1993) The benchmark handbook, 2nd edn. Morgan Kaufmann, San Mateo, California

[GrWa89] Graefe G, Ward K (1989) Dynamic query execution plans. In: Proc ACM SIGMOD Conf, May, Portland, Oregon

[HaSw92] Haas P, Swami A (1992) Sequential sampling procedures for query size estimation. Proc ACM SIGMOD Conf, June, San Diego, California

[HoEn95] Hobbs L, England K (1995) Rdb: a comprehensive guide. Digital Press, Boston, Massachusetts, pp 157–174

[INSS92] Ioannidis YE, Ng RT, Shim K, Sellis TK (1992) Parametric query optimization. In: Proc 18th VLDB Conf, August, Vancouver, Canada

[IoCh91] Ioannidis Y, Christodoulakis S (1991) On the propagation of errors in the size of join results. In: Proc ACM SIGMOD Conf, June, Denver, Colorado

[LiNS90] Lipton R, Naughton J, Schneider D (1990) Practical selectivity estimation through adaptive sampling. In: Proc ACM SIGMOD Conf, June, Atlantic City, New Jersey

[MHWC90] Mohan C, Haderle D, Wang Y, Cheng J (1990) Single table access using multiple indexes: optimization, execution, and concurrency control techniques. Advances in Database Technology EDBT'90, Venice, Italy, pp 29–43

[OlRo89] Olken F, Rotem D (1989) Random sampling from B+ Trees. In: Proc 15th VLDB Conf, Amsterdam, The Netherlands

[Ren92] Rengarajan TK (1992) Rdb/VMS support for multi-media databases, (foils only). In: Proc ACM SIGMOD Conf, San Diego, May

[Roy91] Roy S (1991) Adaptive methods in parallel databases. PhD Dissertation, Department of Computer Science, Stanford University, California

[SACL79] Selinger P, Astrahan M, Chamberlin D, Lorie R, Price T (1979) Access path selection in a relational database management system. In: Proc ACM SIGMOD Conf, Boston, June

[Zipf49] Zipf GK (1949) Human behavior and the principle of least effort. Addison-Wesley, Reading, Massachusetts