Abstract: Block optimization is an attempt to avoid optimizing blocks or parts of a query separately. This process may involve integrating views, derived tables, and subqueries with the rest of the query. For those query blocks that cannot be integrated with the rest of the query, the Teradata optimizer tries to simplify and optimize these blocks. Such optimizations can be achieved using the satisfiability test (checking if a set of conditions are satisfiable) and generating transitive closure, which allows pushing constraints into and out of query blocks.
# Table of Contents

1. **Introduction** .................................................................................................................. 2  
2. **Description of Features** ................................................................................................. 4  
   2.1. Partial group by optimization ..................................................................................... 4  
   2.2. Use check constraints in query optimization ............................................................... 6  
   2.3. Check if a set of constraints is satisfiable ..................................................................... 7  
   2.4. Constraints transitive closure ..................................................................................... 8  
   2.5. Two algorithms for SAT and TC ............................................................................... 9  
   2.6. Eliminate joins based on Referential Integrity (RI) .................................................. 10  
   2.7. Optimize handling of derived tables the same way as views ..................................... 12  
3. **Summary** ....................................................................................................................... 14  
4. **Glossary** ....................................................................................................................... 15  
5. **References** ................................................................................................................... 16  

**Appendix A: Sample Database** ....................................................................................... 17
1. Introduction

Block optimization or BO in Teradata (called de-blocking in IBM DB2 terminology) is concerned about optimizing queries as a whole and not isolating different pieces (blocks) from the rest of the query. Query blocks refer to views, derived tables, subqueries or joins prior to an aggregate. The optimizer chooses block isolation (opposite of BO) because the block is either complex or impossible to integrate with other pieces of the query. We refer to block integration as BI.

There are several BI techniques applied in Teradata prior to V2R5. Examples are view integration and subquery un-nesting. In V2R5, a new BI technique is applied to aggregate queries where the optimizer attempts to apply aggregation prior to joins. Moving the aggregation fully or partially before joins could do that. This optimization technique will be discussed in section 2.1.

Another crucial aspect of BO is to try to push constraints into and out of query blocks and use integrity constraints (like check and referential constraints) for query optimization. We refer to these optimizations as constraints processing or CO. Most of the features in this book fall into this category.

The rest of this document is organized as follows:

- Chapter 2 is the main chapter and it has a description of all the BO features.
  - The partial group by optimization is described in section 2.1 and, as mentioned before, the main purpose is to try to apply early aggregations prior to joins.
  - Section 2.2 discusses the use of check constraints in query optimization and horizontal partitioning.
  - The satisfiability test, which deals with checking if a set of SQL conditions is contradictory, is discussed in section 2.3.
  - Transitive closure solutions are shown in section 2.4. This feature deals with deriving new conditions from the query.
conditions. These new conditions may provide a better execution plans for the query. A special case of transitive closure where new conditions are derived across query blocks is listed in section 2.5.

- Join elimination based on RI is listed in section 2.6. This feature is based on the fact that under certain conditions a join between a parent & child tables could be eliminated. Vertical partitioning of tables is shown as an application of this feature.

- Finally, section 2.7 address derived table’s optimization. The feature basically will guarantee that all views optimizations are applied to derived tables.

- Chapter 3 summarizes the features in this book.
- Chapter 4 is the glossary
- Appendix A shows the DDL of the tables used in the examples.

Note that most of the examples in this book refers to the database defined in Appendix A. The database is similar to the TPC-H database where customers make orders and orders consists of line items. A line item consists of a quantity of a product provided by suppliers. Horizontal partitioning based on order date (o_orderdate) is done to the ordertbl by using check constraints. The partitioning is made by month and therefore there are 12 tables that cover the orders table. The tables are order1, order2, through order12. A view called ordertbl represents all the orders and it is defined as a union of the 12 tables. See Appendix A for a complete definition.
2. Description of Features

2.1. Partial group by optimization

DBMS systems typically execute database operations in the same order as is specified in the query. However, relational algebra provides the freedom to execute these operations in any order as long as the mathematical rules of algebra are strictly followed. For example even though a query implicitly specifies a join order by virtue of how it lists the joins, most database systems would pick a join order that may be different and that is considered optimal for that DBMS implementation.

Similarly when a query specifies joins and aggregation, the normal method is to perform the aggregation on the join result. In some cases this operation can be reversed. Such transformation can significantly benefit complex queries with aggregation and joins. It is outside the scope of this document to discuss when the transformation is appropriate but intuitively it can be performed if the join does not change the number of rows per group over which aggregation is performed.

The idea of performing aggregation before join(s) is to reduce the number of rows being joined. For example, consider two large tables - sales with ten billion rows and inventory with 100 million rows. In order to determine the cost of sales and inventory by state, it may be necessary to join these two tables and aggregate by state. The billion row joins can be replaced with a 50-row join if each of the tables can first be aggregated individually. Please note that the transformation may not always be advantageous, for example when the aggregation does not substantially reduce the input cardinalities to the join. Therefore, the Teradata technique is based on a cost model.

We refer to this transformation as "partial group by". The following example illustrates this concept. A new technique for performing the aggregation has also been added to further improve the transformation. Instead of treating aggregation as a separate operation aggregation, it could be performed, where appropriate, as part of the sort operation.

Consider the following query on the partsupp and lineitem tables.

Q: SELECT l_partkey, SUM(l_quantity),
SUM(ps_supplycost)
FROM partsupp, lineitem
WHERE l_partkey = ps_partkey
GROUP BY 1;

The normal method would be to redistribute lineitem on the join column (l_partkey), perform the join with partsupp and then perform the aggregations on the result of the join. With a TPC-H scale factor of 1000 (i.e., a six billion rows lineitem) this will redistribute 6 billion rows and perform the join with partsupp to produce a 6 billion-row result set on which the specified aggregation operations will be performed.

With "partial early group by" this same query will be performed as follows. For purposes of explanation, views are used to denote the partial results with aggregation even though Teradata does not implement them using views.

CREATE VIEW S1
   (ps_partkey, sumsupcost, count1) AS
SELECT ps_partkey, SUM (ps_supplycost),
   count(*)
FROM partsupp
GROUP BY 1;

CREATE VIEW S2
   (l_partkey, sumlqty, count2) AS
SELECT l_partkey, SUM (l_quantity),
   count(*)
FROM lineitem
GROUP BY 1;

Q': SELECT l_partkey, sumlqty*count1,
sumsupcost*count2

FROM S1, S2

WHERE l_partkey = ps_partkey;

Views S1 and S2 will be materialized first into a spool. The materialized spools S1 and S2 will be joined together, as shown in Q', to get the final result set. Note that S1 would be performed using local aggregation. S2 will be performed as part of the redistribute and sort process required for the join.

The partial-group-by transformation will provide significant performance improvement for this query. Without the transformation, approximately 85 billion rows will be read and written. This is largely because the join of the 6 billion-row lineitem table is joined to the 800 million-row partsupp table to produce a 24 billion-row join result spool for aggregation. With the transformation, the lineitem table is aggregated as part of the sort/redistribution operation to produce a 200 million-row spool. The partsupp table is locally aggregated to produce a 200 million-row spool also. These are joined to produce a 200 million-row result. Overall, there is a reduction of about 3X in the number of rows read and written.

In summary, the optimizer does partial group by optimization automatically when it is cost effective. However, users are encouraged to collect statistics on the group by columns and the join columns. These statistics enable the optimizer to determine the cost effectiveness of partial group by.

2.2. Use check constraints in query optimization

Currently, the optimizer does not use or include check constraints on a table as part of the query plan that access that table. This is because semantically these constraints are redundant while accessing the table since all tuples added to the table were verified to satisfy these constraints. However, in some cases adding these constraints to the query allows the optimizer to find more optimal execution plans. For example, assume that a user would like to list all orders made in the first three months. Assuming that the user has access only to the ORDERTBL view, the query will be like this:

select * from ORDERTBL where extract(month,o_orderdate) <= 3;
For this query, the DBS will access all the tables orders1, ... orders12 with the constraint \( \text{extract(month,o_orderdate) } \leq 3 \) where in reality it only needs to access orders1, orders2 and orders3. The only way the optimizer will filter out the other nine tables is to add the check constraints for every table to the query and to figure out the contradiction between the check constraint and the query constraint. For example, if the check constraint on order4 is added to the corresponding step, the optimizer will have \( \text{extract(month,o_orderdate) } \leq 3 \text{ and extract(month,o_orderdate) } = 4 \) which is a contradiction. For this example the optimizer can simply eliminate this step. As noted in this example the optimizer needs to know if a set of constraints is satisfiable. Such intelligence is missing in the optimizer prior to V2R5. This new feature is discussed in the next section.

Users should be aware of the cost of enforcing check constraints if they decide to use them for horizontal partitioning. Such tradeoffs should be considered by the DBA. Finally, the use of check constraints in query optimization is done automatically by the optimizer and does not require user intervention.

Also, tables can be horizontally partitioned be defining them as partitioned primary index tables (PPI). The details of this V2R5 new structure are outside the scope of this book. However, PPI should be the first choice for horizontal partitioning. Check constraints should be considered in those cases where PPI support is not possible.

### 2.3. Check if a set of constraints is satisfiable

So far there was no need in the Teradata optimizer to check if a set of constraints are contradictory (the opposite is called satisfiable) regardless of the data. The reason is that it will be very rare for a user to submit a contradictory query that does not return results regardless of the data, e.g., a query with a where clause \( a = 1 \) and \( a = 2 \). The example in the previous section shows the need for such checks. Such a problem is called the satisfiability problem or \( \text{SAT} \). Any solution to \( \text{SAT} \) will get a set of constraints and either declare FALSE to denote that they are contradictory and TRUE which means for some specific data the set of constraints are satisfiable.

There are two more applications to \( \text{SAT} \) in the usage and maintenance of materialized views or \( \text{MAT} \). The first problem in \( \text{MAT} \) is to find if a join index needs maintenance for a maintenance operation on one of
its base tables. This problem can be solved by calling SAT for the conjunction of the MAT conditions and the condition applied in the base table maintenance. For example, the operation:

\[ \text{delete lineitem where extract(month,l_shipdate) = 12;} \]

will imply that there is a need to do maintenance for J2 and not for J1. This decision can be made since SAT will return TRUE for “\(\text{extract(month,l_shipdate) = 12 and extract(month,l_shipdate) \geq 7}\)” and FALSE for “\(\text{extract(month,l_shipdate) = 12 and extract(month,l_shipdate) \leq 6}\)”.

Note that such complex conditions in the join index are currently not allowed and it will be one of the new features for MAT in V2R5.

The problem of checking if a join index covers a query or part of the query could be solved in general as a number of SAT problems. The exact relationship between SAT and join index coverage is outside the scope of this document. Note that the use of SAT in this problem is more important once more complex conditions are allowed in the join index definition.

This satisfiability feature is done automatically by the optimizer and does not require user intervention. When applied, certain queries or maintenance operations may run faster. Also, explains may look different due to eliminating parts of a union or due to simplifications of query conditions. Also, the optimization made to join/hash index maintenance may make the use of such indexes more attractive.

### 2.4. Constraints transitive closure

Transitive closure or TC of a set of constraints S1 denoted by TC (S1) is the set of all possible derivable constraints from S1. For example, if S1 is (a=b and a=1) then TC (S1) will be (b=1). Currently, the Teradata optimizer finds transitive closure but is limited to the simple cases like the previous example. It finds TC (S1) only if S1 is a conjunction of constraints and it only derives new ones for a sequence of equality constraints.

In many DSS and CRM applications, TC is needed for date ranges and IN clause and therefore there is a need to extend the current TC implementation to cover these cases. The following example illustrates one of these cases:
SELECT L_SHIPMODE, SUM (CASE WHEN O_ORDERPRIORITY = '1URGENT' OR O_ORDERPRIORITY = '2-HIGH' THEN 1 ELSE 0 END)

From the example above one can find the sequence of <= as S1=(L_SHIPDATE <= L_COMMITDATE-1 and L_COMMITDATE <= L_RECEIPTDATE-1 and L_RECEIPTDATE <= '1994-06-05'). The new set constraints that can be derived from S1 or TC(S1) is (L_COMMITDATE <= ‘1994-06-04’ and L_SHIPDATE <= ‘1994-06-03’). If LINEITEM or one of its join/cover indexes is value ordered/partitioned\(^1\) on L_SHIPDATE then the new constraint L_SHIPDATE <= ‘1994-06-03’ will allow the DBS to access only a partition of the table instead of doing a full table scan.

Similar to the satisfiability test, transitive closure is computed and applied to queries automatically by the optimizer. Users may notice performance improvements for some queries due to these extra conditions. These extra conditions will also show up in query explains.

2.5. Two algorithms for SAT and TC

The problem of SAT and TC are inherently related. For example, we could figure out that (a=1) and (a=2) is contradictory from TC{(a=1) and (a=2)} = {(1=2)}. Another example, TC {(a >= 2 and a <=b and b <= 1} = {b>=2 and a<=1 and 2<=1} which has the contradiction of (2<=1). The two previous examples suggest that SAT is actually a by-product of TC.

There are a lot of algorithms in the research to solve TC and SAT. One of the famous algorithms was done by Rosenkrantz and Hunt [2] which is based on the shortest path algorithm [1]. This algorithm fits well in the scope of conditions we are interested in. This algorithm considers only conjunctive conditions, where each condition is of the form (X op Y + C) or (X op C). Both X and Y are integer variables and C is an integer constant and op ∈ {<,=,>,≥,≤}. In V2R5, this algorithm is implemented to solve TC and SAT with some modification to suit the real domain as well. Another enhancement was made to the above algorithm is to handle ≠.

\(^1\)Currently, Teradata supports value-ordered join indexes. Base table value partitioning is part of a V2R5 feature called Partitioned Primary Index or PPI.
The IN clause is commonly used in customer queries. Also, MAT in V2R5 will allow the IN clause in join index definition. In V2R5, a solution was made to a limited form of conditions that include the IN clause. The conditions are assumed to be conjunctive and each comparison is one of the following forms:

I. \((X=Y)\)

II. \(X \text{ IN} \ (\text{value}_1, \text{value}_2, \ldots), \) where all \(\text{value}_1, \text{value}_2, \ldots \text{valuen} \) are constants.

III. \(X \text{ op} \ \text{Constant}, \) where \(\text{op} \in \{<, =, >, \geq, \leq\}\)

Note that both algorithms work strictly for variables and not for functions on variables. The exception will be the date extract functions. They will be treated like variables in both algorithms. Also, if the conditions has a constraint "C1" on a date column and another condition has a constraint on an extract of that same column, then an extract constraint is added using C1. For example, for the set of conditions \(\{\text{O_orderdate} = '1999-05-01' \text{ and extract(month from O_orderdate) > 2}\}, \) we will add \(\text{extract(month from O_orderdate) = 5 based on O_orderdate = '1999-05-01'}\). Another function will be handled by both algorithms is “IS NULL” and “NOT IS NULL”. Both functions are not applicable to TC but are applicable to SAT. Both algorithms will just return false for SAT if they encounter X IS NULL and NOT X IS NULL. Note that both algorithms are independent of whether the variables are nullable or not.

We also applied the two previous algorithms to cross-query transitive closure, i.e., transitive closure between outer and inner query blocks. This allows pushing conditions into and out of subqueries. The basic approach is to combine the query blocks conditions before computing the transitive closure. The IN and NOT IN clauses are handled as = and \(\neq\). The derived conditions are added appropriately to each block.

2.6. Eliminate joins based on Referential Integrity (RI)

Primary-foreign key relationships between tables are very common in normalized data models and their corresponding physical databases. We will denote these tables as PK-tables and FK-tables. In Teradata,
these relationships can be explicitly defined by the user and are called referential integrity (RI) constraints.

It also common to query PK-tables and FK-tables tables with joins based on the primary and foreign key columns (PK-FK joins). In some of these cases the PK-FK joins are redundant. For example, if the query does not reference columns from the PK-table other than PK-columns itself. Recognizing and eliminating such redundant joins could significantly reduce the query execution time. Such intelligence in the optimizer previously did not exist.

Another area where users use RI is vertical partitioning of a table. Vertical partitioning means that a table with too many columns is split into two tables where one has the “frequently accessed” columns and the other has the other columns. To insure consistency, an RI is needed between the two tables. The choice of PK-table and FK-table in this case is arbitrary but it is advisable to have the table with the frequently accessed columns as the FK-table for reasons that will be shown through the next example. To hide this artificial partitioning, a view is defined that selects all the columns from both tables with a PK-FK join. When a query is submitted on the view that only references columns from the FK-table, the join in the view is obviously redundant and should be eliminated. Note that, without the intervention of the optimizer, the user has no way of removing the join while it is possible to do that in the first case discussed above.

The following query illustrates redundant joins.

```
SELECT S_SUPPKEY, S_ADDRESS, N_NATIONKEY
FROM SUPPLIER,NATION WHERE S_NATIONKEY=N_NATIONKEY
ORDER BY N_NATIONKEY;
```

In the previous query, the join is redundant since every row in SUPPLIER will have exactly one match in NATION (from the RI definition). Also, only N_NATIONKEY is referenced in the query, which can be replaced by S_NATIONKEY. The modified query after join elimination looks like this:

```
SELECT S_SUPPKEY, S_ADDRESS, S_NATIONKEY
FROM SUPPLIER
ORDER BY S_NATIONKEY;
```
The following conditions are sufficient to eliminate a PK-FK join (and of course the PK-table) and are checked by this feature:

- There is an RI defined between the two tables.
- The query conditions are conjunctive.
- No columns from the PK-table other than the PK-columns are referenced in the query. That includes the SELECT, WHERE, GROUP BY, HAVING, ORDER BY, etc.
- The PK-columns in the WHERE clause can only appear in PK=FK joins.

If the above conditions are met then the PK table and PK join will be removed from the query and all references to the PK columns in the query will be mapped to the corresponding FK columns. Also, “NOT NULL” condition on the FK columns will be added if they are nullable. Overall, this optimization will be applied as a rule and no costing will be necessary. Also, the optimizer will do these optimizations automatically. To trigger this optimization, users need to define a RI between the child and parent tables. The cost of RI maintenance could exceed the benefit of join elimination. To alleviate that cost, two new types of RI are introduced in V2R5 called soft and batch RI's. No RI enforcement is made to soft RI's and they are used mainly as a hint for join elimination. Soft RI's should be used only when the user is confident that the parent-child relationship is consistent. Batch RI's are kind of a middle ground between soft and regular RI's. They are enforceable but the cost to do that is usually less than that of regular RI's. For more details about the new RI types, the reader is referred to the V2R5 documentation.

2.7. Optimize handling of derived tables the same way as views

Since the introduction of derived tables in Teradata, the optimizer always materializes them prior to any other operations on these derived tables. In short, BI or CI was not done for derived tables.

Semantically, derived tables are just like views since both are logical definition of some SQL query. However, there are lots of attempts to optimize handling views by the optimizer through BI in some cases where the view is integrated with other blocks of the query or by some
**CI** optimizations for aggregate views and views defined by a union. The list of **CI** and **BI** optimizations for views is beyond the scope of this document but the main issue here is that the optimizer should apply all these optimizations to derived tables as well.

The way to implement this optimization is not to repeat the view **BI** optimizations to derived tables. Rather, at an early stage in the parser we will convert the tree structure of a derived table to a pseudo-view. The pseudo-view definition is easily computable from the derived table definition. All references to that derived table reference will be replaced by references to the pseudo-view, which also will carry the view definition as a sub-node. During the resolver phase the view definition will be found as part of parser tree and not from the dictionary or the dictionary cache.

Users in the past used derived tables to force a specific join order. With the optimizations described in this document and other optimizations added in V2R5, the need to force a specific join order is significantly reduced. If a problem does occur and it becomes necessary to force a specific join order, a dbscontrol flag may be set to enable the old behavior of derived tables. Such a problem should be reported so that it can be determined why the optimizer is not choosing an optimal join order automatically. It is not recommended to enable this behavior except as a temporary workaround since this may cause the optimizer not to find the optimal plan in other cases.
3. Summary

Five block optimization techniques are implemented in V2R5. Overall, all these optimizations are done automatically by the optimizer and do not require user intervention. Applications like horizontal or vertical partitioning will require the creation of check and referential integrity constraints. The user should consider the tradeoffs between the benefits of these optimizations and the costs to maintain these constraints. Among the five optimizations, only derived tables optimization can be turned off. We do not advise the user to do that unless it is really necessary as a work around for a non-optimal join order. Finally, the following table summarizes the optimizations in this book, their applications, user intervention and tradeoffs.

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Automatic</th>
<th>Application</th>
<th>Tradeoffs</th>
<th>“Turn off” option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partial Group By</td>
<td>Yes</td>
<td>NA</td>
<td>NA</td>
<td>NO</td>
</tr>
<tr>
<td>Use of Check Constraints</td>
<td>Yes</td>
<td>Horizontal Partitioning</td>
<td>Cost of constraints maintenance</td>
<td>NO</td>
</tr>
<tr>
<td>Satisfiability &amp; Transitive Closure</td>
<td>Yes</td>
<td>NA</td>
<td>NA</td>
<td>NO</td>
</tr>
<tr>
<td>Join Elimination</td>
<td>Yes</td>
<td>Vertical Partitioning</td>
<td>Cost of RI maintenance</td>
<td>NO</td>
</tr>
<tr>
<td>Derived Tables Optimization</td>
<td>Yes</td>
<td>NA</td>
<td>NA</td>
<td>YES</td>
</tr>
</tbody>
</table>
4. Glossary

**DBS** – Acronym for Database Software, that is, the software making up the Teradata RDBMS that is concerned with the management and processing of the databases in a Teradata RDBMS and the SQL requests concerning those databases.

**DDL** – Acronym for data definition language. One of the two major divisions of Teradata SQL statements (the other being DML). Includes statements such as CREATE, DROP, etc.

**SAT** – The problem of testing if a set of constraints is satisfiable or not.

**TC** – Acronym for Transitive closure.

**RI** - Acronym for referential integrity.

**MV** - Acronym for materialized views.

**BI** - Acronym for Block integration.

**CO** - Acronym for Constraints optimization.

**MAT** - Acronym for materialized views.
5. References


[2] Processing conjunctive predicates and queries. VLDB 80, pages 64-72, D. Rosenkrantz and H. I. Hunt

[3] Teradata RDBMS V2R5.0, Performance Optimization
Appendix A: Sample Database

CREATE SET TABLE order1 ,NO FALLBACK ,
NO BEFORE JOURNAL,
NO AFTER JOURNAL
(
  O_ORDERKEY INTEGER NOT NULL,
  O_CUSTKEY INTEGER NOT NULL,
  O_ORDERSTATUS CHAR(1) CASESPECIFIC NOT NULL,
  O_TOTALPRICE DECIMAL(15,2) NOT NULL,
  O_ORDERDATE DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  O_ORDERPRIORITY CHAR(15) CASESPECIFIC NOT NULL,
  O_CLERK CHAR(15) CASESPECIFIC NOT NULL,
  O_SHIPPRIORITY INTEGER NOT NULL,
  O_COMMENT VARCHAR(79) CASESPECIFIC NOT NULL,
CHECK (EXTRACT(MONTH FROM O_ORDERDATE) = 1))
UNIQUE PRIMARY INDEX ( O_ORDERKEY );

CREATE SET TABLE order2 ,NO FALLBACK ,
NO BEFORE JOURNAL,
NO AFTER JOURNAL
(
  O_ORDERKEY INTEGER NOT NULL,
  O_CUSTKEY INTEGER NOT NULL,
  O_ORDERSTATUS CHAR(1) CASESPECIFIC NOT NULL,
  O_TOTALPRICE DECIMAL(15,2) NOT NULL,
  O_ORDERDATE DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  O_ORDERPRIORITY CHAR(15) CASESPECIFIC NOT NULL,
  O_CLERK CHAR(15) CASESPECIFIC NOT NULL,
  O_SHIPPRIORITY INTEGER NOT NULL,
  O_COMMENT VARCHAR(79) CASESPECIFIC NOT NULL,
CHECK (EXTRACT(MONTH FROM O_ORDERDATE) = 2))
UNIQUE PRIMARY INDEX ( O_ORDERKEY );

CREATE SET TABLE order12 ,NO FALLBACK ,
NO BEFORE JOURNAL,
NO AFTER JOURNAL
(
  O_ORDERKEY INTEGER NOT NULL,
  O_CUSTKEY INTEGER NOT NULL,
  O_ORDERSTATUS CHAR(1) CASESPECIFIC NOT NULL,
  O_TOTALPRICE DECIMAL(15,2) NOT NULL,
  O_ORDERDATE DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  O_ORDERPRIORITY CHAR(15) CASESPECIFIC NOT NULL,
  O_CLERK CHAR(15) CASESPECIFIC NOT NULL,
  O_SHIPPRIORITY INTEGER NOT NULL,
  O_COMMENT VARCHAR(79) CASESPECIFIC NOT NULL,
CHECK (EXTRACT(MONTH FROM O_ORDERDATE) = 12))
UNIQUE PRIMARY INDEX ( O_ORDERKEY );
CREATE VIEW ORDERTBL\(^2\) AS
SELECT * FROM ORDER1 UNION SELECT * FROM ORDER2 UNION
SELECT * FROM ORDER3 UNION SELECT * FROM ORDER4 UNION
SELECT * FROM ORDER5 UNION SELECT * FROM ORDER6 UNION
SELECT * FROM ORDER7 UNION SELECT * FROM ORDER8 UNION
SELECT * FROM ORDER9 UNION SELECT * FROM ORDER10 UNION
SELECT * FROM ORDER11 UNION SELECT * FROM ORDER12;

CREATE SET TABLE lineitem, NO FALLBACK,
NO BEFORE JOURNAL,
NO AFTER JOURNAL
( 
  L_ORDERKEY INTEGER NOT NULL,
  L_PARTKEY INTEGER NOT NULL,
  L_SUPPKEY INTEGER NOT NULL,
  L_LINENUMBER INTEGER NOT NULL,
  L_QUANTITY DECIMAL(15,2) NOT NULL,
  L_EXTENDEDPRICE DECIMAL(15,2) NOT NULL,
  L_DISCOUNT DECIMAL(15,2) NOT NULL,
  L_TAX DECIMAL(15,2) NOT NULL,
  L_RETURNFLAG CHAR(1) CASESPECIFIC NOT NULL,
  L_LINESTATUS CHAR(1) CASESPECIFIC NOT NULL,
  L_SHIPDATE DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  L_COMMITDATE DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  L_RECEIPTDATE DATE FORMAT 'yyyy-mm-dd' NOT NULL,
  L_SHIPINSTRUCT CHAR(25) CASESPECIFIC NOT NULL,
  L_SHIPMODE CHAR(10) CASESPECIFIC NOT NULL,
  L_COMMENT VARCHAR(44) CASESPECIFIC NOT NULL)
PRIMARY INDEX (L_ORDERKEY);

CREATE JOIN INDEX J1 AS SELECT * from LINEITEM where extract
(month, l_shipdate) <= 6;

CREATE JOIN INDEX J2 AS SELECT * from LINEITEM where ex-
tract(month, l_shipdate) >= 7;

CREATE JOIN INDEX LIAgr AS SELECT SUM(L_QUANTITY), L_ORDERKEY
FROM LINEITEM GROUP BY L_ORDERKEY;

CREATE VIEW LIQTY AS SELECT SUM(L_QUANTITY), L_ORDERKEY
FROM LINEITEM GROUP BY L_ORDERKEY;

CREATE SET TABLE NATION, NO FALLBACK,
NO BEFORE JOURNAL,
NO AFTER JOURNAL
( 
  N_NATIONKEY INTEGER NOT NULL UNIQUE,
  N_NAME CHAR(26) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,
  N_REGIONKEY INTEGER NOT NULL,
  N_COMMENT VARCHAR(116) CHARACTER SET LATIN NOT CASESPECIFIC NOT
  NULL)
UNIQUE PRIMARY INDEX (N_NAME);

\(^2\) Note that UNION of the tables in the view has the drawback that uniqueness of o_orderkey can no longer
be enforced by the DBS. That is, currently no capability to have unique index on a view of a union.
CREATE SET TABLE supplier ,NO FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL
    ( 
        S_SUPPKEY INTEGER NOT NULL,
        S_NAME CHAR(26) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,
        S_ADDRESS VARCHAR(41) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,
        S_NATIONKEY INTEGER NOT NULL,
        S_PHONE CHAR(15) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,
        S_ACCTBAL DECIMAL(13,2) NOT NULL,
        S_COMMENT VARCHAR(102) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,
        FOREIGN KEY ( S_NATIONKEY ) REFERENCES NATION ( N_NATIONKEY ))
    PRIMARY INDEX ( S_SUPPKEY );

CREATE SET TABLE partsupp ,NO FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL
    ( 
        PS_PARTKEY INTEGER NOT NULL,
        PS_SUPPKEY INTEGER NOT NULL,
        PS_AVAILQTY INTEGER NOT NULL,
        PS_SUPPLYCOST DECIMAL(13,2) NOT NULL,
        PS_COMMENT VARCHAR(199) NOT CASESPECIFIC NOT NULL
    )
    PRIMARY INDEX ( PS_PARTKEY )
    INDEX ( PS_SUPPKEY );