

Transbase®
Query Optimization and Locking Guide

Transaction Software GmbH
Willy-Brandt-Allee 2
D-81829 München
Germany
Phone: +49-89-62709-0
Fax: +49-89-62709-11
Email: info@transaction.de
<http://www.transaction.de>

Version 6.8.1.40
November 02, 2010

Contents

1	Introduction	3
2	Transactions and Locks	4
2.1	Lock Modes	5
2.1.1	Update Locks and Exclusive Locks	6
2.1.2	Lock Granularity: Page Locks and Table Locks	7
2.1.3	Transbase [®] Strategy of Lock Granularity	7
2.2	Limitations of Concurrency under Page Locks	9
2.3	Consistency Levels	9
2.3.1	Consistency Level 3	9
2.3.2	Consistency Level 2	10
2.3.3	Consistency Level 1	10
2.3.4	Overview of Consistency Levels	11
2.4	Manual Table Locking	11
3	Query Optimizer	12
3.1	General Introduction to Operator Trees	12
3.2	B-Trees and Hypercube Trees	15
3.3	The Sample Database	15
3.4	B-Tree Key Access in Single Table Queries	15
3.4.1	B-Tree Key Access, Pointsets and Intervals	16
3.4.2	Single Key Predicates on B-Trees	16
3.4.3	Pointsets on B-Trees	18
3.4.4	Key Intervals and Interval Sets on B-Trees	19

3.4.5	B-Tree Key Fields in Expressions	21
3.4.6	B-Trees with Compound Key	22
3.4.7	Pointsets and Intervals on Compound Keys (B-Trees)	23
3.4.8	Compound Keys and Intervals (B-Trees)	23
3.4.9	Query Schemata which are non Optimizable on B-Trees	24
3.5	Predicates with OR on B-Trees	25
3.6	Pattern Matching	26
3.7	Secondary Indexes	26
3.7.1	Predicate Evaluation on Secondary Indexes	27
3.7.2	Tuple Materialization	29
3.7.3	How the Optimizer Chooses an Index	29
3.7.4	How to Override the Optimizer's Choice	30
3.7.5	Multiple Secondary Indexes	31
3.7.6	Suppressing the Tuple Materialization	32
3.7.7	Using Secondary Indexes to Reduce I/O	34
3.7.8	Overhead of Secondary Indexes in INSERT, UPDATE, DELETE	35
3.8	Sort Orders of Single Table Queries	35
3.9	The Hypercube Tree	36
3.9.1	Syntax for Creation of a Hypercube Table	37
3.9.2	Performance Behaviour of a Hypercube Table	37
3.9.3	Number and Order of Hypercube Keys	38
3.9.4	Symmetry of Hypercube keys	38
3.9.5	Comparison of B-Trees with Hypercube Trees	39
3.9.6	Hypercube Trees as Secondary Indexes	39
3.10	Join Queries	39
3.10.1	Terms	39
3.10.1.1	Join (Equi-Join)	39
3.10.1.2	Local Restrictions	40
3.10.1.3	Sort Merge Join	41
3.10.1.4	Nested Loop Join	43
3.10.2	The Join Optimization Rule	45

3.10.2.1	Examples for Join Queries	45
3.10.3	Multi-Way Joins	47
3.10.4	User Control of Transbase Join Strategy	51
3.10.4.1	Keywords NLJOIN and SMJOIN	51
3.10.4.2	NLJOIN and SMJOIN in Multi-Way Join Queries	52
3.10.4.3	Setting the Joinmode with Tbmode	53
3.10.4.4	Risks of Explicitly Specified Join Methods	54
3.10.4.5	Hints for Join Query Optimization	55
3.10.4.6	Outer Joins	56
3.10.5	Aspects of Multi-Column Joins	56
3.11	Subqueries Delivering a Value	57
3.12	Subqueries with IN and EXISTS	59
3.13	Evaluation of UNION, INTERSECT, DIFF	60
3.14	Queries with NOT IN and NOT EXISTS	61
3.15	Queries with GROUP BY and DISTINCT	62
3.16	INSERT, UPDATE, DELETE Queries	64
3.16.1	INSERT Queries	64
3.16.2	DELETE Queries	65
3.16.3	UPDATE Queries	65
3.16.4	Modification Queries with Self Reference	65
3.17	The Data Spooler	66
3.18	GREEDY Mode for DISTINCT and UNION	67
3.19	Appendix	67

Chapter 1

Introduction

The performance of an application may be insufficient due to various reasons.

- The runtime of specific queries may be poor.
- Waiting times for locks may be large.
- Locking overhead may be too large due to unsuitable locking granularity.

This manual addresses the issues of locking (chapter 2) and query optimization (chapter 3). The latter does not include a guide for schema development but gives the necessary background for understanding the performance behaviour of a given query under a given schema.

Chapter 2

Transactions and Locks

A transaction is defined as a sequence of database statements which preserves data consistency and data security.

Transactions may either be committed or aborted. For this purpose, at the TBX interface 2 calls named CT and AT are supported. In ESQL, the statements are termed COMMIT WORK and ROLLBACK WORK.

A COMMIT operation makes all changes of a transaction permanent, whereas an ABORT operation undoes all changes of the calling transaction.

Consider an accounting program which has to preserve the balance. A certain amount has to be added to one account and has to be subtracted from another account. This is realized as a sequence of two update statements. However, if a failure occurs between the first and the second statement, the database is in an inconsistent state and thus not correct.

By the notion of transaction, those two update actions may be concentrated in a single unit and thus are made atomic: It will be guaranteed that in case of a failure either both operations are done or none of the operations is done. Anyway, the database will be left in a consistent state. Data consistency also means that a transaction sees a consistent state of the data even in case of other concurrent transactions which even might change data which was seen by other transactions. This property is sometimes described as data security.

Since a database system should schedule concurrent transactions fully parallel, another problem might violate data consistency. Consider a transaction T_c counting the tuples of a table and a concurrent transaction T_i inserting records into the same table. For consistency reasons, the counting transaction has to see a database state which either reflects *no inserted tuples* or *all inserted tuples*, i.e. T_c has to be scheduled before T_i or after T_i . However, this would result in sequentialization of transactions which would degrade concurrency totally.

Instead, Transbase[®] schedules the read and write actions of transactions so that the overall result of all transactions is the same as if they had been executed serially. This property is called serializability.

Transbase[®] guarantees **serializability by default**. Serializability is also often called Consistency Level 3. Its effect is that (apart from waiting for resources and for locks) each transaction sees the database as if it ran alone on the database.

Transbase[®] additionally offers lower consistency levels which yield higher concurrency at the expense of logical soundness. They are described later in this section.

To achieve serializability, transactions have lock objects before accessing them. Objects may be locked and unlocked either implicitly (by SQL statements) or explicitly (by LOCK / UNLOCK statements). Since locks are not always compatible with other locks, locking restricts concurrency for the sake of consistency.

The following chapters describe lock modes and lock granularities.

2.1 Lock Modes

Transbase[®] distinguishes three kinds of locks, namely read locks, update locks, and exclusive locks. The latter 2 lock modes both serve to update data in the database: whereas exclusive locks allow for no concurrency, update locks allow concurrency of a single write transaction and many read transactions, and read locks allow for full concurrency. Note that in nearly all other database systems no difference is made between update locks and exclusive locks, so that concurrency is more heavily restricted than in Transbase[®].

The compatibility of locks can be described by the following compatibility matrix, where a "+" means that locks are compatible and a "-" means that locks are not compatible on the same object.

	<i>read</i>	<i>update</i>	<i>exclusive</i>
read	+	+	-
update	+	-	-
exclusive	-	-	-

If a transaction requests a lock on an object which is locked by another transaction in non-compatible mode, the transaction has to wait until the other transaction frees its lock. Two problems result from waiting: First, a transaction may wait indefinitely (since the other transaction lasts very long) or, secondly, transactions wait for one another (this situation is well-known as deadlock).

Transbase[®] takes care of both situations: Indefinite delay is prevented by timeouts which may be specified by application programs. If a transaction runs into timeout

it is not aborted, but simply returns an error flag to the application program with the meaning "locks have not been granted within timeout period". The application program is free to decide whether to abort the transaction or to try again (perhaps with a longer timeout interval).

Deadlocks are detected by Transbase[®] very efficiently:

Local: As long as a transaction works only local on a single database, a graph is held where a deadlock is represented by a cycle. If a cycle is detected, the transaction which caused the cycle is aborted. Using graphs maximizes concurrency .

Global: If a transaction gets distributed over more databases, the transaction receives a dynamic timestamp whose value is derived from the graph. One can say that the timestamp contains part of the graph information in condensed form (The details of the algorithm are not described here).

Even global deadlocks are securely detected by this mechanism. However, some situations may be detected as deadlocks that would not be considered so if one would have a global distributed graph (which would be very costly to maintain). In other words, the property of maximal concurrency is left when a transaction gets distributed .

2.1.1 Update Locks and Exclusive Locks

As shown in the lock compatibility diagram, Transbase[®] provides an update lock for write transactions and thus increases concurrency between readers and one writer on the same table. To understand the difference between update locks and exclusive locks, we have a short look at the implementation. The atomic behaviour of transactions is established by storing the state of database objects to disk before they are changed by the active transaction. The changes of the active transaction are made on a copy (sometimes called a shadow or simply a new "version"). A subsequent COMMIT operation discards the old version and makes the new version available to others transactions, whereas an ABORT operation discards the new version and reinstalls the old version .

While a writing transaction is active, the old version can be made available to readers since the old version reflects a consistent database state (namely the state before start of the writing transaction). Thinking in terms of serializability , this means that the reading transactions have to be scheduled as if they had been executed before the writing transaction. Those reading transactions are sometimes called "old readers".

To guarantee serializability , Transbase[®] records those dependencies in another graph, which is actually integrated with the wait graph described above.

To commit a writing transaction, it is necessary to wait for old readers to disappear. In terms of locks, the update lock has to be converted into an exclusive lock at COMMIT point. Note that this conversion may cause delay, but never will cause a deadlock .

2.1.2 Lock Granularity: Page Locks and Table Locks

The objects that are locked by Transbase[®] are tables or pages. None of both locking strategies is superior to the other one in all circumstances.

Properties, Pros and Cons of table locking:

Prop: All tables accessed by a transaction are locked statically at the end of the compilation phase. No locks are requested during the evaluation phase.

Pro: Scanning large parts of a table requires just one lock , not many locks on finer granularity, this results in small locking overhead .

Pro: No phantom problems occur.

Con: No or limited concurrency on one table in case of many writers or one writer and readers.

Properties, Pros and Cons of page locking:

Prop: Exactly the pages which are effectively accessed are locked (at query evaluation time).

Pro: Several updaters on a table may concurrently run if they access different parts of the table.

Con: Performance is decreased to an extent which ranges from slightly (if the query accesses few pages) until considerable (if page locks heavily accumulate).

2.1.3 Transbase[®] Strategy of Lock Granularity

Assume a statement S where a table T occurs: Transbase[®] uses a table lock on table T in the following seven cases (T1) until (T7):

(T1) T is one of the system tables.

(T2) T occurs in one of the following DDL statements: CREATE INDEX, DROP INDEX, ALTER TABLE, DROP TABLE, SPOOL TABLE FROM <file>.

- (T3) T is subject of a LOCK statement.
- (T4) T occurs in an arbitrary statement and the user-definable lock strategy has been set to TABLES by a "TBMODE LOCKMODE TABLES" (see TBMODE statement).

For the remaining 3 cases we additionally assume that the user-definable lock strategy has not been set to PAGES by a "TBMODE LOCKMODE PAGES" (see TBMODE statement).

- (T5) The database is a CD Editorial or CD Retrieval Database.
- (T6) The statement is a SELECT (not FOR UPDATE) or DELETE or UPDATE and there is at least one WHERE clause with a search condition referring to T where not all values of the single or compound key are specified with an explicit or implicit equality (i.e. it is not guaranteed that at most one tuple is fetched from T or deleted from T or updated in T, resp.).
- (T7) The statement is a INSERT INTO T SELECT .. and there is at least one table in the SELECT expression which is table locked according to rule (T6).

In all other cases, Transbase[®] uses page locks.

For better understanding and for complementary reasons we now informally describe the cases where Transbase[®] sets page locks:

The following three conditions must all be fulfilled for a page lock on table T:

- (PC1) T is not one of the system tables
- (PC2) The statement is a SELECT, INSERT, UPDATE or DELETE or SPOOL INTO file SELECT ..
- (PC3) If the database is a CD Editorial or a CD Retrieval database, then the user must have defined TBMODE LOCKMODE PAGES (see TBMODE statement).

In addition to all above conditions, one of the following cases must hold for page lock:

- (P1) The user must have defined TBMODE LOCKMODE PAGES (see TBMODE statement).
- (P2) The statement is a SELECT or DELETE or UPDATE statement and all search conditions on T must have specified all key values with equality.
- (P3) The statement is a SELECT .. FOR UPDATE statement.
- (P4) The statement is a INSERT INTO T VALUES statement or a INSERT statement with SELECT clause where conditions in (P2) apply.

Brief Summary: On CD Editorial and CD Retrieval databases, table locks are preferred because updates are assumed to be rare or not subject to concurrency. In all statements, page locks are used if the query can syntactically be analysed as a single tuple query (`SELECT .. FOR UPDATE` is an exception). All Transbase[®] decisions can be overridden by the `TBMODE LOCKMODE` statements.

2.2 Limitations of Concurrency under Page Locks

Page locks can be used if maximal concurrency is needed - conflicts between writers or between readers and writers then can only occur if the transactions access the same page.

However, beside the pages of the primary index of a table, there are additional pages where transactions might conflict:

- Pages of secondary indexes
- Pages for internal keys (IK pages)

Recall that IK pages exist if the `CREATE TABLE` statement creation did not specify `WITHOUT IKACCESS`. IK values are necessary for secondary indexes.

Both kinds of pages concentrate information of many tuples in one page and are thus more likely for conflicts than primary index pages. They tend to be hot spots for concurrency even under page locks.

Only if the table has been created `WITHOUT IKACCESS`, it is guaranteed that transactions accessing tuples in different pages do not conflict.

2.3 Consistency Levels

Transbase[®] supports 3 consistency levels. For short, these levels are called `CONS3`, `CONS2` and `CONS1`. Each transaction runs in one of the 3 consistency levels. It can be changed between any 2 transactions inside the application.

2.3.1 Consistency Level 3

The highest consistency level (`CONS_3`) guarantees serializability of transactions by implementing so called 2-phase-locking. Before a table is accessed (read or written) an appropriate lock is requested automatically. This lock will be held until the end of the transaction.

Recall that all locks are requested and set automatically by Transbase[®].

2.3.2 Consistency Level 2

The second consistency level (`CONS_2`) shows no difference to `CONS_3` for write operations, i.e. update locks are automatically set and held until EOT. However, read locks are held only while the query is active. With consistency level 2 it can no longer be guaranteed that repeated reads deliver the same results within the same transaction: Read reproducibility is not given.

On the other hand, consistency level 2 allows higher concurrency since read locks are released earlier. This means that updaters of a table can commit while the reading transaction is still active (however the reading query must have finished already).

2.3.3 Consistency Level 1

Even if read transactions run with consistency level 2, it may happen that updaters of a table are hindered in committing by read transactions if the read transactions run very long read queries. This can be avoided by running the read transactions in the lowest consistency level 1 (`CONS_1`). Read queries in level `CONS_1` run without any locks. While a read query in `CONS_1` is active, updaters of the read data may commit. It is not predictable whether the read query sees the changed data in a whole or partially or not at all.

For example, if a read query counts the number of tuples in a table `T` and an updater increases the number of tuples in `T` from 1000 to 1013 and commits before the read query has finished, the read query may produce any result between 1000 and 1013. Thus, with `CONS_1`, results of read queries may be inconsistent. However, `CONS_1` is very useful to run lengthy statistical queries on tables without hindering the update traffic on the same tables (actually only the request "DROP TABLE `T`" of an update transaction would lead to blocking of the update transaction at commit time as long as a `CONS_1` read query on `T` is still active).

A `Cons1` transaction also may run update queries. As far as update queries are concerned, `CONS_1` behaves much like `CONS_2` and `CONS_3`. In detail, an update query in `CONS_1` sets locks on the updated tables like `CONS_2` and `CONS_3` but no locks on tables which are only read by the update query. For example, one could run a long statistical read query on a table `T` and insert the result into a table `S` by the query:

```
INSERT INTO S
SELECT COUNT(*), AVG(t1), MAX(t2) FROM T
```

If this query were run in `CONS_1` mode, a write lock until end of transaction would be placed on `S` but no locks would be placed on `T`.

2.3.4 Overview of Consistency Levels

The characteristics of the three consistency levels are shown below in the diagram.

	CONS_1	CONS_2	CONS_3
duration of read locks	none	query	transaction
serializability and read reproducibility	no	no	yes
query consistency	no	yes	yes
concurrency	highest	medium	lowest

Table 2.1: Consistency Levels

The consistency level can be set dynamically by an application; the default setting is CONS_3, i.e. highest consistency.

The explicit consistency choice is possible when no transaction is active:

(TBX) `tbx(SET_CONSISTENCY, CONS_x)`

(ESQL) `EXEC SQL SET CONSISTENCY CONS_x`

Note: Lock granularity and consistency level do not depend on each other.

2.4 Manual Table Locking

Instead of locking implicitly by Transbase[®], also explicit locking with `LOCK` and `UNLOCK` statements is possible. An application program may lock objects earlier or stronger than Transbase[®] would do automatically. Conversely, the application might release locks earlier than Transbase[®] would. Note that by default locks are held until the end of a transaction in order to guarantee full serializability .

Manual locking may be used e.g. to implement a hierarchical locking scheme which is guaranteed to be deadlock -free. Another reason for manual locking might be the prevention of indefinite delay at commit point, when an update lock has to be converted to an exclusive lock .

Releasing locks before the end of a transaction leads to a loss of data consistency but may increase concurrency . Releasing a read lock before the end of transaction might produce so-called non-reproducible reads. This means that repeated read accesses to the same table might produce different results, namely if an update transaction on this table has been committed meanwhile.

Note that update locks cannot be manually released.

Chapter 3

Query Optimizer

The Transbase[®] Relational Database System offers the TB/SQL query language for data retrieval and manipulation. TB/SQL is a superset of the ISO 9075 standardized ISO-SQL language. As SQL is a high level descriptive language, queries formulated in this language are automatically optimized by the Transbase[®] query optimizer.

This manual describes to a far extent, how TB/SQL queries are optimized. For each query, the optimizer generates an execution plan (also called access plan) which is either stored or immediately executed. If more than one execution plan is possible, the optimizer tries to find the best one. We first describe operator trees in general in a separate section and then discuss particular operator trees for several query types.

Additionally, this manual describes how the optimizer can be influenced by query formulations as well as by explicit control mechanisms.

3.1 General Introduction to Operator Trees

Like most other database management systems, Transbase[®] uses operator trees, in order to execute queries. A query represented as SQL statement is compiled by the Transbase[®] compiler and transformed in an unoptimized operator tree. This operator tree can be executed (interpreted), but is optimized by the Transbase[®] optimizer before execution.

The optimizer uses schema information, in order to determine the most efficient operator tree. Note that Transbase[®] has a rule-based optimizer. Thus, statistics about table and predicate cardinality usually do not influence the optimizer.

After the operator tree has been optimized, it is executed by the Transbase[®] interpreter. Generally, the interpreter starts from the leaves of the tree and executes the tree in a pipelined manner. To understand an operator tree, you should look

at the leaves and follow the paths to the root. Each node denotes an operation (as label of the node) .

In Transbase[®], the operator tree is represented as text structure. To understand this structure, some practice is necessary. We shall show some examples and discuss the most important nodes. The understanding of complex operator trees is easier when converting the text representation into a graphical tree.

The operator tree itself can be determined via the *tbi* interactive interface tool and via programming interfaces *TBX* and *TCI*. For example, execute the following commands in *tbi*:

```
SET PLANS ON
select * from inventory where description='WASHER';
```

The plan is stored in the file plan.txt in the current directory. Here is the content of the file:

```
select * from inventory where description='WASHER'

(N0:sel {  updatable   3 "partno" "description" "qonhand"  }
                                           --#tup: 0
  (N1:restr  --#tup: 0
    (N2:rel  { "inventory"   proj 3(1 2 3) } --#tup: 0
      )
    (N3:eq   { }
      (N4:attr { N1[2] } )
      (N5:const { 'washer' char(6) } ))))
{nosort}

(N0:sel {  updatable   3 "partno" "description" "qonhand"  }
                                           --#tup: 1
  (N1:restr  --#tup: 1
    (N2:rel  { "inventory"   proj 3(1 2 3) } --#tup: 9
      )
    (N3:eq   { }
      (N4:attr { N1[2] } )
      (N5:const { 'washer' char(6) } ))))
{nosort}
```

First, the query itself is written. Then, two operator trees are shown. The first operator tree is written to the file before the query is executed (after the optimizing step). The second operator tree is written after the query has been executed. It

is enriched by additional information about the actual number of tuples processed by each node.

Each node of the operator tree is denoted by *N* with a unique number. Thus, a node can be referenced by other nodes using for example *N5*. Each node itself stores a tuple. The attributes of the tuple are referenced by *N5[1]* for the first attribute. Note that the attribute positions start with 1.

The above operator tree shows that a full table scan is executed on the table *inventory* (node *N2*). This full table scan returns 9 tuples (tuple counter is 9). The *rel* node (denoting the full table scan) is a son of the above *restr* node which performs the filtering of the restriction. The predicate is an equity operation (second son of the *restr* node) on the second attribute of the tuples coming into the *restr* node *N1*. Since the *rel* node returns the complete tuple, the tuples of the *restr* node correspond to the tuples from the relation *inventory*.

The *sel* node shows the names of the result columns (attribtues) which correspond to the attribute names of the *inventory* table. The number of tuples fulfilling the *restr* node is 1. Thus, the query returns 1 tuple (see tuple counter of the *sel* node).

The `{nosort}` directive is for internal information of the database kernel indicating that the tree is correct w.r.t. sorting. We further omit it.

The following query shows a key access:

```
select name, address from suppliers where suppno=54;
```

```
(N0:sel { updatable 2 "name" "address" } --#tup: 1
  (N1:rel { "suppliers" proj 2(2 3) pgllocks } --#tup: 1
    (N2:ivmk { eq } --#tup: 1
      (N3:const { 54 integer } ))))
```

The above operator tree contains a key access to the relation *suppliers* and the restriction is an equity comparison with the value *54*. Note that the node *ivmk* denotes the key restriction for the base relation (in this case *suppliers*). *ivmk* is a node representing an interval. The property of the interval is indicated by the parameter in parantheses (in this operator tree *eq*, i.e., an equity comparison).

Most nodes in the operator tree have one son, but there are also nodes with more sons (e.g., a *times* node representing a join has always two sons, a *build* node can have any number of sons). The parantheses `')'` identify the structure of the tree uniquely.

We do not provide a full description of all types of nodes here. Some are described in the following sections.

3.2 B-Trees and Hypercube Trees

All table data and secondary index data is stored in standard B-Trees or so called Hypercube Trees. In either case, all user data are stored in the leaf nodes. The key access information is stored in the index nodes. In general, the index nodes do not contain the full keys but only minimal separator information.

B-Trees and Hypercube Trees differ in some important aspects which will be described in a special chapter. In the following, we concentrate on standard B-Trees.

Three access methods are supported by standard B-Trees.

The *sequential access method* traverses tuples one by one and delivers the tuples in ascending or descending key sort order. This access method is also called a *sequential scan*.

The *key access method* has a key value as argument and traverses the B-Tree from the root to one of the leaves. As the height of a B-Tree typically is 3 or 4 and almost never more than 5, this method is very fast.

The *IK access method* is an optional access method strongly related to the usage of secondary index trees which are created by the `CREATE INDEX` statement. A secondary index is also a B-Tree the key of which is the field or field combination specified in the `CREATE INDEX` statement. Each secondary index tuple contains an identifier (internal key, IK) of the corresponding base tuple in the base table B-Tree. Given an IK, the corresponding tuple can be directly accessed via an IK mapping table. The IO and CPU overhead is comparable to a key access.

Often, the key access method is combined with the sequential access method. First the tree is traversed using a given key value and then, starting with the found tuple if any, the tree is traversed sequentially to the end or until the key values exceed a given second key value. Here the processing time of course depends on the scanned tuple range on the B-Tree leaves.

3.3 The Sample Database

The sample database used in most of the examples is depicted in the Appendix.

Note that the combination (`suppno`, `partno`) is a compound key in table quotations, the two other tables have a non-compound key.

3.4 B-Tree Key Access in Single Table Queries

A single table query is of the form

```

SELECT ti, tj, ..
FROM   t
WHERE  < P(t) >

```

The **FROM** clause contains exactly one table name *t*. The **SELECT** clause contains * or fields of *t* or expressions containing field names, constants or host variables. The **WHERE** clause if any is a boolean expression (predicate) containing field names of *t* or constants or host variables, this is schematically denoted by $P(t)$.

For simplicity, during this chapter it is assumed that no subquery is in the **SELECT** clause or **WHERE** clause.

Example:

```

SELECT name, address
FROM   suppliers
WHERE  suppno = 54

```

One possible access plan which always works is the sequential access method. Here, the B-Tree containing table *t* is scanned sequentially and for each tuple it is tested if it fulfills the predicate.

This access plan, however, is slow if the table is large. In the following we discuss under which circumstances key access methods can be applied by the optimizer.

3.4.1 B-Tree Key Access, Pointsets and Intervals

This chapter describes in which cases the optimizer can exploit the key access method. Of course, this depends on the predicate in the **WHERE**-Clause.

As notational convenience, key fields are always denoted *key1*, *key2*, etc. In a compound key, *key1* denotes the highest weighted key.

The notation *x1*, *x2* etc. stands for expressions where no field names occur. For example, *x1* may be a constant, a host variable or an expression composed of constants or host variables.

3.4.2 Single Key Predicates on B-Trees

As a first case, consider the following predicate scheme:

```
key1 = x1
```

Example:

```
SELECT name, address
FROM   suppliers
WHERE  suppno = 54
```

```
(N0:sel { updatable 2 "name" "address" } --#tup: 1
  (N1:rel { "suppliers" proj 2(2 3) pglocks } --#tup: 1
    (N2:ivmk { eq } --#tup: 1
      (N3:const { 54 integer } ))))
```

The Transbase[®] optimizer generates the key access method here. Note, additionally, that if the key is not compound then the query result has at most one tuple. Anyway, the matching tuples are found very fast even if the table is extremely large. The operator tree also shows the key access (*ivmk* node). The constant for the key is represented by the node *const*.

Additional predicates may be connected with **AND** and involve other operators.

Example:

```
SELECT name, address
FROM   suppliers
WHERE  suppno = 54 AND name LIKE 'Smi%'
```

```
(N0:sel { updatable 2 "name" "address" } --#tup: 0
  (N1:restr --#tup: 0
    (N2:rel { "suppliers" proj 2(2 3) pglocks } --#tup: 1
      (N3:ivmk { eq } --#tup: 1
        (N4:const { 54 integer } )))
    (N5:like { false }
      (N6:attr { N1[1] } )
      (N7:const { 'Smi%' char(4) } ))))
```

Here the optimizer again provides for the highly performant key access via *suppno*, the resulting tuple (if any) is then tested on the additional predicate (this overhead is not measurable).

In the operator tree, there is a *restr* node standing for the additional restriction. In this case, the **AND** operator of the SQL statement does not occur in the operator tree, because the primary restriction (key access) is evaluated via direct key search (as son of the *rel* node and the restriction with the *ivmk* subtree) and the additional restriction is evaluated only for tuples already fulfilling the key restriction.

In the above operator tree, the key restriction returns one tuple which does not fulfill the LIKE predicate. Thus, there is no tuple delivered from the `restr` node (tuple counter is 0) and no tuple from the `sel` node.

The `like` node is an example for the attribute reference. A like operation is performed for the first attribute of each tuple of node `N1`, denoted by `N1[1]`.

Note that the above query comprises an AND-connection. OR-connections between different fields behave completely different, this is discussed in Chapter "Predicates with OR".

3.4.3 Pointsets on B-Trees

The predicate scheme

```
key1 = x1
```

can be generalized to

```
key1 = x1 OR key1 = x2
```

Example:

```
SELECT name, address
FROM   suppliers
WHERE  suppno = 54 or suppno = 57
```

```
(N0:sel { updatable 2 "name" "address" } --#tup: 2
  (N1:rel { "suppliers" proj 2(2 3) pglocks } --#tup: 2
    (N2:ivuni --#tup: 2
      (N3:ivmk { eq } --#tup: 1
        (N4:const { 54 integer } ))
      (N5:ivmk { eq } --#tup: 1
        (N6:const { 57 integer } ))))))
```

This operator tree shows that there is a key access to the relation `suppliers` consisting of two point searches in the B-Tree. The restriction is a union of two point searches, denoted by the node `ivuni`. `ivuni` has two children (`ivmk` nodes with the corresponding values for the restriction).

An equivalent formulation is possible with the `IN` list operator:

```
key1 IN (x1 , x2 , x3 , ..)
```

Example: (with additional predicate)

```
SELECT name, address
FROM   suppliers
WHERE  suppno IN (54 , 57 , 17) AND name like 'TITAN%'
```

With the above query schemata, the optimizer generates an access plan which makes n key accesses ($n \geq 1$) where n depends on the number of specified key values. The alternative query formulations with OR and with the IN list behave completely identically.

The above query schema can be considered as a specification of several points in the range of the key - it therefore is called a pointset. Also the trivial case of only one single point can be thought of as a pointset.

As only key accesses are involved, the performance of the pointset query class is also very high. Additionally, caching effects further reduce the processing time.

Note also that the result tuples are produced in sort order corresponding to the B-Tree key even if the key values in the IN list are not sorted. The key values are sorted at runtime before the tree access is done.

3.4.4 Key Intervals and Interval Sets on B-Trees

Now key intervals are considered. Intervals result from one of the following predicate schemata:

```
key1 > x1
key1 < x1
key1 >= x1
key1 <= x1
```

```
key1 > x1 AND key1 < x2
```

```
key1 BETWEEN x1 AND x2
```

Example: (with additional predicate)

```
SELECT name, address
FROM   suppliers
WHERE  suppno BETWEEN 54 AND 57
AND   name like 'TITAN%'
```

```

(N0:sel { updatable 2 "name" "address" } --#tup: 2
  (N1:restr --#tup: 1
    (N2:rel { "suppliers" proj 2(2 3) } --#tup: 2
      (N3:ivsec --#tup: 1
        (N4:ivmk { ge } --#tup: 1
          (N5:const { 54 integer } ))
          (N6:ivmk { le } --#tup: 1
            (N7:const { 57 integer } )))))
      (N8:like { false }
        (N9:attr { N1[1] } )
        (N10:const { 'TITAN%' char(6) } ))))

```

In all the above schemata, a range of key values is specified, i.e. an interval. In some cases, the interval is one-sided infinite. The optimizer recognizes key intervals and generates the following access plan: one key access is performed with the left interval boundary, then a sequential scan is performed until a tuple is found whose key value exceeds the right interval boundary. These steps are represented in the operator tree by the nodes `ivsec` and the two children `ivmk` nodes, one with "greater or equal 54" (`ge`) and the other with "lower or equal 57" (`le`). For all tuples found on this scan, the predicates (if any) which are additionally specified are evaluated.

Of course, the evaluation time of such a query depends on the size of the key range, but the important fact is that full advantage of the key access method is taken also in this kind of query.

If more than one interval is specified and connected via OR or AND operators, the optimal key access is still done by the optimizer:

Example: (with additional predicate)

```

SELECT name, address
FROM suppliers
WHERE (suppno < 30
      OR
      suppno BETWEEN 54 AND 57
      OR
      suppno > 113)
AND name like 'TITAN%'

```

```

(N0:sel { updatable 2 "name" "address" } --#tup: 1
  (N1:restr --#tup: 1
    (N2:rel { "suppliers" proj 2(2 3) } --#tup: 2
      (N3:ivuni --#tup: 3

```

```

(N4:ivuni  --#tup: 2
  (N5:ivmk { lt  }  --#tup: 1
    (N6:const { 30 integer } ))
  (N7:ivsec  --#tup: 1
    (N8:ivmk { ge  }  --#tup: 1
      (N9:const { 54 integer } ))
    (N10:ivmk { le  }  --#tup: 1
      (N11:const { 57 integer } ))))
(N12:ivmk { gt  }  --#tup: 1
  (N13:const { 113 integer } ))))
(N14:like { false }
  (N15:attr { N1[1] } )
  (N16:const { 'TITAN%' char(6) } ))))

```

Here three intervals are specified. The optimizer generates an access plan which three times performs a key access (first is trivial) followed by a scan until the right interval boundary.

Note also that the result tuples again arrive in ascending key order independent of the interval order in the query. This means that the intervals are sorted at runtime and also merged if they happen to overlap. For example, if the left boundary of the third interval happens to lie inside the second interval then the last two intervals in effect are treated as one interval with extended boundary.

Combinations of points, pointsets and intervals work equally, i.e. a point is simply treated as a degenerated case of an interval.

3.4.5 B-Tree Key Fields in Expressions

The formulation `key1 = x1` is the most usual one but it is also possible and *harmless* to specify one of the following:

`x1 = key1`

`key1 + x1 = x2`

`key1 - x1 = x2`

Additions and subtractions are *harmless* as long as `key1` only appears once in the whole expression. However, multiplications and divisions as well as multiple occurrences of the key field hinder the key access optimization and thus possibly cause poor performance.

Example:

```
key1 + 10 > 2 * key1  -- not optimized !
```

```
key1 * x1 = x2      -- not optimized !
```

3.4.6 B-Trees with Compound Key

Very often, a table has been specified with more than one key component, e.g. the table quotations in the sample database. Such a key is called a compound key. Compound keys often are very useful and even necessary to avoid an artificial and space consuming additional unique key such as serial number.

However, the optimization and key access techniques are slightly more complicated to understand. A very good and helpful example is the telephone directory which also has a compound key consisting of last name (highest key) and first name (lower key). Even the remaining components (address etc.) participate in the compound key but last and first name are sufficient to understand the search techniques.

Assume a compound key with 2 dimensions whose components are named key1 and key2. For example, in table quotations, suppno and partno constitute such a compound key.

All predicate schemata discussed so far (the notation key1 refers to the highest key now) are optimized as in the non-compound case. The only possible difference in the behaviour of the query is the result cardinality: If a single key value is specified (key1 = x1) then in the non-compound case at most one tuple hits, but in the compound case many tuples might hit.

One sees that in the compound case already the simplest key specification (namely a point specification like: key1 = x1) in effect describes an interval on the tuples in the leaves. Thus, the evaluation technique here is again that of an interval: one key access followed by a scan until a tuple is seen whose first key is greater than the specified key value.

Compare this with a search in the telephone book for last name = 'Smith'. The search for the first hit is perfectly fast, the overall time, however, depends on the number of hits.

Now assume that the predicate refers to more than one key component, e.g. the predicate scheme

```
key1 = x1 AND key2 = x2
```

Here the optimizer also generates a perfectly fast access plan, namely a key access with both key values followed by a B-Tree scan until the first tuple is seen whose

key values are different from the specified one (in the 2-dimensional case the scan becomes trivial).

This is comparable to a search for "Smith", "David" in the phone book.

3.4.7 Pointsets and Intervals on Compound Keys (B-Trees)

Remember that a pointset is an OR connection of several values referring to the same key field. Most often this is implicitly specified by a IN formulation such as:

```
key1 IN (x1, x2, ...).
```

Pointsets over compound keys have the following shape:

```
key1 IN (x11, x12, ... x1n) AND
key2 IN (x21, x22, ... x2m)
```

This query scheme is also optimized to multiple key accesses. Semantically, this is a search for $n*m$ key combinations. The query is executed as $n*m$ key accesses each followed by a corresponding table scan.

3.4.8 Compound Keys and Intervals (B-Trees)

Intervals on compound keys are more complicated to understand. Consider the 2-dimensional interval

```
key1 BETWEEN x11 AND x12
AND
key2 BETWEEN x21 AND x22
```

With B-Trees, in this query scheme, the optimizer generates a key access which refers to key1 only!

This means that all tuples are scanned where key1 is between x11 and x12. The additional condition about key2 will only be tested on the scanned tuples but cannot be integrated into the key access.

The effect is that the number of scanned tuples is possibly much larger than the number of hits. Compare this with the search in the phone book for all entries where the last name is between "Mason" and "Moore" and the first name is between "James" and "Jim". The first entry with "Mason" can directly be accessed but all names between "Mason" and "Moore" have to be scanned to find the hits, i.e. it is unfeasible to directly find the first "James" whose first name is bigger than "Mason" because the next occurring last name after "Mason" is not known.

Intervals on more than one key component thus are not optimally supported by B-Trees. However, this is exactly the access pattern for which the so called Hypercube Tree is made for.

Note that if the predicate specifies a pointset on the highest key and an interval on the second key, then B-Trees work perfectly.

```
key1 IN (x11, x12, ..x1n) AND
key2 BETWEEN x21 AND x22
```

Here a key access on the compound key (x11, x21) is generated and the tuple scan runs until (x11, x22). Then a key access with (x12, x21) and a tuple scan until (x12, x22) follows, etc. In summary we have n direct key accesses each followed by a tuple scan. Only those tuples which finally fulfill the key condition are scanned.

Of course, also the scheme

```
key1 = x11 AND
key2 BETWEEN x21 AND x22
```

is perfectly optimized on B-Trees because it is a special case of the above.

For the interested reader, the general case remains to be described. Assume a predicate which refers to the n highest components key1 to keyn of a compound key. Schematically, it is :

```
P1(key1) AND P2(key2) AND ... AND Pn(keyn)
```

Assume that the j-1 first predicate parts are pointsets and Pj is an interval. The optimizer can generate key accesses such that P1 to Pj are evaluated via appropriate key accesses. All remaining predicates Pj+1 to Pn must be evaluated separately, i.e. some tuples are filtered out and more tuples than finally hit are scanned.

3.4.9 Query Schemata which are non Optimizable on B-Trees

If a predicate refers to key1 and to key3 for a compound key (with at least 3 components) then, of course, the key3 part cannot be used for key access.

If a predicate does not refer to the highest key component, then no key access at all is possible (however see Hypercube Tree).

Care must be taken if some predicate parts are connected by the OR operator. This is discussed in the next chapter.

3.5 Predicates with OR on B-Trees

In the many examples given so far, it has often be demonstrated that key access optimization is not affected by additional predicate parts which are connected with AND.

Care must be taken if some predicate parts are connected by the OR operator.

Example:

`key1 = x1 OR field = x2`

Since `field` is a nonkey field, no key access is possible because there may be tuples with `field = x2` but `key1 <> x1` which would not be found if a key access with `x1` were performed.

However, as discussed, in the schema

`key1 = x1 OR key1 = x2`

a sequence of key accesses is possible because both predicate parts refer to the key.

In more complex examples, like

`(key1 = x1 AND field = x2)`
 OR
`(key1 = x3 AND field = x4)`

it is not evident at first sight that key access is still possible. Transbase[®] derives key access by the following algorithm: All predicate parts which do not refer to keys are set to `TRUE` and the resulting predicate is simplified as far as possible. If a non-trivial predicate remains then it can be used as key access.

In the above example, the predicate parts `field = x2` and `field = x4` are replaced by `TRUE`, the remaining predicate is `(key1 = x1 OR key1 = x3)` which can be used as key access.

However, in example 3.5, the predicate part `<field = x2>` is set to `TRUE` and the remaining predicate evaluates to `TRUE` because of the OR operator, and so it is seen that no key access is possible.

Transbase[®] also applies this algorithm for each of the given secondary indexes to check if it is applicable. This is described in chapter 3.7 (Secondary Indexes).

3.6 Pattern Matching

The SQL predicates `LIKE` and `MATCHES` can be optimized in some important cases. Consider the schemata

```
key1 LIKE 'a%z'
```

and

```
key1 MATCHES 'a.*z'
```

Both formulations are semantically equivalent. The pattern here consists of a non-empty prefix "a" followed by a wildcard. Note that "a" here stands for any non-empty string, i.e. we consider patterns like 'data%' or 'Y%' or 'sys%min' but not '%ystem'.

If key1 is again the highest key of a B-Tree (primary index or also secondary index, see the following chapters) then the above pattern evaluation is optimized by an additional key access and restricted tuple scan. The key value is taken from the prefix of the pattern (i.e. "data" in case of pattern "data%") and the tuple scan is finished if the first tuple is seen whose key1 value does not start with "data".

By this technique, the number of scanned tuples can be dramatically decreased if the pattern prefix is selective.

3.7 Secondary Indexes

As described earlier, a primary index B-Tree is created automatically on each user table (unless a Hypercube Tree has explicitly been specified). The key is taken from the `KEY IS` clause or implicitly as the combination of all fields.

Additionally, secondary indexes can be created on fields or field combinations. A secondary index by default is also a B-Tree the key of which is the field or field combination specified in the `CREATE INDEX` statement. In addition to the field value(s) each secondary index tuple contains an identifier (internal key, IK) of the base tuple of the corresponding base table B-Tree. Given an IK, the corresponding tuple can directly be accessed via a IK mapping table which can be thought of as part of the primary B-Tree. The I/O and CPU overhead is comparable to a key access.

Let the table quotations be as depicted in the Appendix. Assume a secondary index be created as

```
CREATE INDEX quot_qon_price ON quotations(qonorder, price)
```

quot_qon		
qonorder	price	IK
0	0.05	14
0	0.08	5
0	0.10	2
0	0.20	12
0	0.25	3
0	18.00	6
0	21.00	10
7	19.50	16
20	29.00	15
24	8.50	11
50	0.30	1
150	0.10	7
200	0.02	9
200	0.04	8
200	0.10	4
200	0.20	13

In the table, the layout of tuples of that secondary index is depicted.

For simplicity, it is assumed here that the IK value is identical to the position of the tuple in the primary index (in reality, this does not hold).

Note that the (compound) key of the secondary index is (qonorder, price) such that the tuples are in another order than the corresponding tuples in the primary index.

3.7.1 Predicate Evaluation on Secondary Indexes

All that has been discussed up to now about key access is also valid for secondary indexes. The terms "key", "highest key", "compound key" etc. directly apply to the keys of secondary indexes.

For example, "qonorder" is (the highest) key in secondary index `quot_qon_price` and the key of `quot_qon_price` is compound - even the IK field is part of the key but this is irrelevant to the user because he or she never searches for that field).

Example:

```
SELECT *
FROM quotations
WHERE qonorder IN (24, 200)
```

Without the secondary index, this query would have to scan the entire quotations table. The IN predicate is postfiltered on all tuples delivered from the full table scan.

```
(N0:sel {  updatable  5 "suppno" "partno" "price" "deliv_time"
          "qonorder"  } --#tup: 5
  (N1:restr  --#tup: 5
    (N2:rel  { "quotations"  proj 5(1 2 3 4 5) } --#tup: 16
      )
    (N3:inls
      (N4:attr { N1[5] } )
      (N5:const { 24  integer } )
      (N6:const { 200  integer } ))))
```

With the secondary index, the optimizer generates an access plan which does 2 key accesses each followed by a scan on the secondary index (see the `index` node in the following operator tree). The query result consists of 5 tuples.

```
(N0:sel {  updatable  5 "suppno" "partno" "price" "deliv_time"
          "qonorder"  } --#tup: 5
  (N1:mat  { "quotations" } --#tup: 5
    (N2:index { "quot_qon_price"  proj 1(3)
              user_tablename="quotations" } --#tup: 5
      (N3:ivuni  --#tup: 2
        (N4:ivmk { eq  } --#tup: 1
          (N5:const { 24  integer } ))
        (N6:ivmk { eq  } --#tup: 1
          (N7:const { 200  integer } )))))))
```

For interval predicates (`>`, `<`, `BETWEEN`, etc.) key accesses are generated as in the primary index case, but note that for a predicate with the `<>` operator, the optimizer never takes a secondary index on behalf of key access.

From the discussion of B-Trees as primary indexes it now should also be clear that the following example can perfectly be processed with the compound secondary index on (qonorder, price):

Example:

```
SELECT *
FROM   quotations
WHERE  qonorder = 0 AND price > 1.50
```

3.7.2 Tuple Materialization

There is an important difference between the usage of primary and secondary indexes. In the latter case the query evaluation plan additionally must access the primary index via the IK value after a matching tuple is found in the secondary index (see the node *mat* in the previous operator tree). This is necessary if the query result needs tuple fields which are not in the secondary index (e.g. as in example 3.7.1, but see also chapter 3.7.6).

Fetching a tuple via the IK value is called tuple materialization (TM). One TM is roughly as expensive as one key access. If the TM is necessary for each result tuple, then for large results the secondary indexes are slower than the primary index, perhaps even slower than a complete sequential scan of the primary index.

3.7.3 How the Optimizer Chooses an Index

In some cases, the primary index as well as a secondary index would be suitable for key access.

Example:

```
SELECT *  
FROM quotations  
WHERE suppno = 54 AND qonorder = 200 AND price > 1.50
```

Two optimized evaluation plans are possible here:

Key access via the primary key (*suppno=54*) and testing the remaining condition (*qonorder=200 AND price > 1.50*) for the resulting 4 tuples,

or

Key access via the secondary index (*qonorder=200 AND price > 1.50*) and testing the remaining condition (*suppno=54*) for the resulting tuples (after tuple materialization).

In the general case, it would be reasonable to take the index whose corresponding predicate parts produce the smallest amount of tuples. The optimizer has no information about the expected number of tuples. Instead, some rather simple rules based on syntactic criteria of queries are used to determine the query plan.

The primary index is always chosen if all its keys are specified whereby the *n-1* highest keys are specified with Pointsets (see 3.4.8). In all other cases, we define

for each index I (primary or secondary) a number $DDA(I) \geq 0$. DDA means "degree of direct access". In general, $DDA(I)$ is the maximum of all numbers d with the property that the d highest weighted keys of I occur in the query and are directly supported by I in the query processing.

In our example the primary key has only one key attribute and the equity predicate ensures that at most one tuple is returned. Thus the optimizer does not consider the DDA and automatically chooses the primary key access.

See section (Multiple Secondary Indexes) how this generalizes to the case of multiple secondary indexes.

3.7.4 How to Override the Optimizer's Choice

It may happen that a query exhibits very poor performance because the optimizer chooses a secondary index which produces very many tuples which must all be materialized (perhaps in contradiction with the computed hit ratio). The query can be reformulated such that a certain index is not used by the query evaluation plan.

The method is to connect a trivial predicate which always yields FALSE (e.g. $1 = 0$) with an OR operator to the predicate part which refers to the undesired index.

For example, in example 3.7.1, if the primary index shall not be used but the secondary index instead, one formulates:

Example:

```
SELECT *
FROM quotations
WHERE (suppno = 54 OR 1 = 0 ) AND qonorder = 200
```

Note the additional parentheses which are absolutely necessary.

The following operator tree is generated from the query without additional $OR\ 1 = 0$. Here a key access to the base relation is performed:

```
(N0:sel { updatable 5 "suppno" "partno" "price" "deliv_time"
          "qonorder" } --#tup: 2
  (N1:restr --#tup: 2
    (N2:rel { "quotations" proj 5(1 2 3 4 5) } --#tup: 4
      (N3:ivmk { eq } --#tup: 1
        (N4:const { 54 integer } )))
    (N5:eq { }
      (N6:attr { N1[5] } )
      (N7:const { 200 integer } ))))
```

This operator tree is generated by the above query forcing the optimizer to chose the secondary index access:

```
(N0:sel {  updatable   5 "suppno" "partno" "price" "deliv_time"
          "qonorder" } --#tup: 2
  (N1:restr  --#tup: 2
    (N2:mat { "quotations" } --#tup: 4
      (N3:index { "quot_qon_price"   proj 1(3)
                 user_tablename="quotations" } --#tup: 4
        (N4:ivmk { eq } --#tup: 1
          (N5:const { 200 integer } )))))
    (N6:or
      (N7:eq { }
        (N8:attr { N1[1] } )
        (N9:const { 54 integer } ))
      (N10:eq { }
        (N11:const { 1 integer } )
        (N12:const { 0 integer } )))))
```

3.7.5 Multiple Secondary Indexes

Here we assume the existence of several secondary indexes on one table for example index quot_qon_price(qonorder, price) as before and additionally index quot_suppno(partno).

Example:

```
SELECT *
FROM quotations
WHERE partno = 221 AND qonorder = 200 AND price > 0.5
```

Both secondary indexes are candidates for processing. The optimizer need not make a choice but creates a query plan where both indexes are used followed by a intersection (based on IK values) of the 2 partial results. This also generalizes on an arbitrary number of secondary indexes.

```
(N0:sel {  updatable   5 "suppno" "partno" "price" "deliv_time"
          "qonorder" } --#tup: 0
  (N1:mat { "quotations" } --#tup: 0
    (N2:merge { 3 proj 1 mpush 1 0 } --#tup: 0
      (N3:set  --#tup: 0
        (N4:build
```

```

      (N5:const { -2147483648 integer } )))
(N6:index { "quot_partno"      proj 1(2)
          user_tablename="quotations" } --#tup: 1
(N7:times {      } --#tup: 1
(N8:ivmk { eq } --#tup: 1
(N9:const { 221 integer } ))
(N10:ivmk { ge } --#tup: 1
(N11:attr { N2[1] } ))))
(N12:sort { +1 , } --#tup: 0
(N13:index { "quot_qon_price"   proj 1(3)
          user_tablename="quotations" } --#tup: 0
(N14:times {      } --#tup: 1
(N15:ivmk { eq } --#tup: 1
(N16:const { 200 integer } ))
(N17:ivmk { gt } --#tup: 1
(N18:const { 0.5 numeric(1,1) } ))))))))

```

The intersection of the indexes is indicated by the `merge` node which has in this example three children. The merge operation itself is quite complicated and not explained here in detail. Note that the `times` node stands for the combination of predicates (**ANDing**) used for the key access to the corresponding index. The `sort` node is necessary to sort the IK values from the second index (`quot_qon_price`) to efficiently intersect them with the IK values of the first index (`quot_partno`). The result tuples from the first index need not be sorted, because they come sorted from the index. After the index merge (index intersection) step, the tuples must be materialized as shown earlier.

Note that there are different combinations for index intersection. However, the basic pattern is similar to the operator tree above and can be recognized when analyzing operation trees.

In contrast to the intersection of several secondary indexes, it is not possible to use primary and secondary indexes followed by an intersection. Thus the optimizer only has to decide between primary index and (one or several) secondary indexes. This is based on the DDA calculation described in the previous section. In case of several usable secondary indexes, the DDA is defined as the maximum of all DDA's of the secondary indexes.

3.7.6 Suppressing the Tuple Materialization

If one single secondary index is used for query processing, the optimizer additionally checks whether the TM via the IK value can be suppressed. This is the case if all fields which are used in the query are also components of the secondary index.

Example:

```
SELECT COUNT(*)
FROM quotations
WHERE qonorder = 200
```

```
(N0:sel { 1 "column_0" } --#tup: 1
  (N1:group { count[*] } --#tup: 1
    (N2:index { "quot_qon_price" proj 1(3)
      user_tablename="quotations" } --#tup: 4
      (N3:ivmk { eq } --#tup: 1
        (N4:const { 200 integer } )))))
```

In this example, no field beyond the `qonorder` is needed, so the TM is suppressed which leads to a very fast execution plan.

In the operator tree, you see the `group` node that is necessary for the `count(*)` operation. This operator groups the resulting tuples, 4 in the sample query, to one tuple and counts their number denoted by `count[*]` as parameter of the `group` node.

Example:

```
SELECT suppno, partno
FROM quotations
WHERE qonorder = 200
```

Obviously, the TM would take place here. If the secondary index `quot_qon_price` had been created on `(qonorder, suppno, partno)` or `(qonorder, partno, suppno)`, then it would also be accessed via (highest) key `qonorder` but the TM could be suppressed. Of course, the price paid would be more space needed for the secondary index. But with the current index `quot_qon_price(qonorder, price)` the following operator tree is generated:

```
(N0:sel { updatable 2 "suppno" "partno" } --#tup: 4
  (N1:proj --#tup: 4
    (N2:mat { "quotations" } --#tup: 4
      (N3:index { "quot_qon_price" proj 1(3)
        user_tablename="quotations" } --#tup: 4
        (N4:ivmk { eq } --#tup: 1
          (N5:const { 200 integer } ))))
      (N6:build
        (N7:attr { N1[1] } )
        (N8:attr { N1[2] } ))))
```

As an extreme but sometimes reasonable strategy, one could make a secondary index which contains all fields (but of course with a different key than the primary index). This would allow very efficient access via an alternative "key" without any overhead of TM, but of course the additional disk space is considerable.

This shows that the user's choice for secondary indexes sometimes is a difficult one because a trade off between runtime of queries and needed disk space must be taken into account. Also update overhead must be considered - this is described in chapter 3.7.8.

3.7.7 Using Secondary Indexes to Reduce I/O

Secondary indexes sometimes are used by the optimizer even if the predicate does not lead to key accesses on them.

Example:

```
SELECT COUNT(*)
FROM quotations
```

```
(N0:sel { 1 "column_0" } --#tup: 1
  (N1:group { count[*] } --#tup: 1
    (N2:index { "quot_partno" proj 1(2)
              user_tablename="quotations" } --#tup: 16
    )))
```

Clearly this query does not exhibit any key access opportunities. However, the query optimizer would prefer a secondary index because it is much smaller and thus incurs less I/O.

In general, if there is no predicate exhibiting a key access opportunity, the optimizer takes the smallest (in tuple length) index which contains all needed fields (no tuple materialization). If there is a key access opportunity, then the index choice is made solely on the basis of key access opportunities. Of course, the tuple materialization is suppressed if the chosen index has all fields needed by the query. This has already been described in chapter 3.7.6 and is also an important database design criterion as far as the choice of secondary indexes is concerned. In other words, only if no key access is possible then the index choice is made based on the needed fields. The strategy is that the savings gained by key access (reduction of number of tuples) are bigger than those gained by the choice of a smallest index (reduction of size of tuples).

3.7.8 Overhead of Secondary Indexes in INSERT, UPDATE, DELETE

Each secondary index may have positive effect on query response time for some query schemata but it causes overhead in disk space and update maintenance.

Each insertion or deletion of a tuple into the primary index triggers insertion or deletion of a tuple into *each* secondary index. For UPDATE queries, not all secondary indexes must be maintained but only those which contain a field that occurs in the SET clause of the UPDATE query. For such a updated field, one tuple deletion and one insertion takes place in the secondary index.

If very large updates, deletions and/or insertions have to be done on the base table, it may even be more economic to drop secondary indexes and to recreate them after the modifications have been finished.

The choice for secondary indexes will primarily be based on the required direct access. It should carefully be designed which fields to make additional members of the index to save tuple materialization (this is described in the preceding chapter). The additional runtime overhead in index maintenance may be neglectible, the additional space overhead may well pay off.

3.8 Sort Orders of Single Table Queries

As described in the previous chapters, for a single table query either the primary or one of the secondary indexes (with or without tuple materialization) is used. Independent of the key accesses and the values of key accesses, the result tuples are always ascendingly sorted according to the key of the used index.

If an explicit sort order is specified in the query (ORDER BY clause) the optimizer checks whether the result tuples must be explicitly sorted according to the specified order or whether the key order of the used index already fulfills the required sort order.

Example:

```
SELECT *
FROM quotations
WHERE delivery_time > 20
ORDER BY suppno
```

```
(N0:sel { 5 "suppno" "partno" "price" "deliv_time"
          "qonorder" } --#tup: 0
(N1:restr --#tup: 8
(N2:rel { "quotations" proj 5(1 2 3 4 5) } --#tup: 16
```

```

)
(N3:gt { }
  (N4:attr { N1[4] } )
  (N5:const { 20 integer } ))))

```

or

```
ORDER BY suppno, partno
```

or

```
ORDER BY suppno desc
```

```

(N0:sel { 5 "suppno" "partno" "price" "deliv_time"
         "qonorder" } --#tup: 0
  (N1:restr --#tup: 8
    (N2:rel { "quotations" desc proj 5(1 2 3 4 5) } --#tup: 16
      )
    (N3:gt { }
      (N4:attr { N1[4] } )
      (N5:const { 20 integer } ))))

```

Note that in this operator tree, there is also no explicit sort operator, since the B-Tree can be read backwards. This is indicated by the parameter `desc` in the `rel` node N2.

or

```
ORDER BY suppno desc , partno desc
```

This is the same operator tree as above. It can be read backward again because for both B-Tree key attributes the `DESC` keyword is specified.

In all examples the optimizer recognizes that no explicit sorting must be performed at runtime. For descending orders it has to be noted that Transbase[®] B-Trees also can be scanned backwards, also if key accesses are involved. Therefore, the optimizer generates backward scans if descending orders are specified. Therefore, no additional overhead is produced by `ORDER BY` clauses which are consistent with the key sort order in the above sense.

3.9 The Hypercube Tree

Each table and each secondary index can be created as a standard B-Tree (single key or compound key) or as a Hypercube Tree.

3.9.1 Syntax for Creation of a Hypercube Table

The syntax for creation of Hypercube Trees is very similar to that of a standard B-Tree. An example for creation of a Hypercube table:

```
CREATE TABLE geopoints (
  longitude NUMERIC(10,7) NOT NULL
             CHECK(longitude BETWEEN -180 AND 180),
  altitude  NUMERIC(10,7) NOT NULL
             CHECK(altitude BETWEEN -90 AND 90),
  pointinfo INTEGER )
HCKEY IS longitude, altitude;
```

The clause HCKEY is the syntactic indication for a Hypercube Tree. Check constraints indicating a range for the values are mandatory for each key and, for performance reasons, should not be chosen much larger than necessary.

In contrast to the keys of a standard B-Tree, the key types for a Hypercube Tree are restricted to the arithmetic types TINYINT, SMALLINT, INTEGER, BIGINT, NUMERIC(p,n) and additionally DATETIME.

3.9.2 Performance Behaviour of a Hypercube Table

In many aspects, a Hypercube table behaves differently to a B-Tree table. The most striking property is the optimal support of multidimensional range queries. For example:

```
SELECT pointinfo
FROM geopoints
WHERE longitude BETWEEN 81.500 AND 89.523
AND altitude BETWEEN 50.123 AND 53.12
```

```
(N0:sel { updatable 1 "pointinfo" } --#tup: 0
  (N1:rel { "geopoints" proj 1(3) } --#pages(index/leaf/nohit):
    0/0/0 --#tup: 0
    (N2:times { } --#tup: 1
      (N3:ivsec --#tup: 1
        (N4:ivmk { ge } --#tup: 1
          (N5:const { 81.500 numeric(5,3) } ))
          (N6:ivmk { le } --#tup: 1
            (N7:const { 89.523 numeric(5,3) } )))
        (N8:ivsec --#tup: 1
          (N9:ivmk { ge } --#tup: 1
```

```

(N10:const { 50.123 numeric(5,3) } ))
(N11:ivmk { le } --#tup: 1
(N12:const { 53.12 numeric(4,2) } ))))))

```

This query specifies a 2-dimensional query (querybox) or a 2 dimensional subcube of the data space. A standard B-Tree with compound key on (longitude,altitude) only can use the interval on longitude for efficient restriction of I/O for the result tuples whereas the condition on altitude must be postfiltered. This means that very many tuples not lying in the result are fetched from disk.

The Hypercube table, however, uses a multidimensional clustering and thus arranges disk blocks in a way that guarantees a very high average percentage of hits per disk block for most multidimensional interval queries.

The operator tree for that query shows the interval generation for the multidimensional query box for both dimensions. This two dimensional query box is used for the key access to the Hypercube table `GEOPPOINTS`. Note that the intervals can be specified via much more complex operations (e.g., joins with other tables, sub queries etc.). This is the typical case for data warehouses. Note that the `rel` node for Hypercube trees is enriched by additional information for statistical analysis. The number of accessed index and leaf pages is reported as well as information about how many contained data for the result set.

3.9.3 Number and Order of Hypercube Keys

With a standard B-Tree, the number of specified keys is not a very critical performance parameter (as opposed to the order of keys). In other words, if key1 to keyn are specified as keys and a certain query on these keys is processed in a certain execution time then the tail extension of this compound key with additional fields does not change the execution time. In contrast, a Hypercube Tree needs a careful decision for the number of keys (but NOT for the order of keys). The reason is that the performance mainly depends on the ratio of searched keys to Hypercube keys. For example, a Hypercube table with 3 keys is searched by queries Q1, Q2, Q3 where each Qi searches i of the 3 keys, for i = 1, 2, 3. Then Q1, Q2, Q3 will tend to exhibit increasing performance in this order. Performance here means number of result tuples per time unit. The reasonable upper limit of number of Hypercube keys depends on the table size but in any case is about 5 or 6.

3.9.4 Symmetry of Hypercube keys

In contrast to a standard B-Tree, the ordering of keys of a Hypercube table is not relevant. The index structure is symmetrical. For some applications, one single Hypercube table thus can replace several secondary B-Tree indexes on a B-Tree table.

3.9.5 Comparison of B-Trees with Hypercube Trees

None of the two index structures B-Tree and Hypercube Tree is superior to the other for all queries. The Hypercube strongly outperforms the B-Tree for multidimensional queryboxes where most of the keys are specified. With decreasing number of searched keys the performance gain decreases. The B-Tree outperforms the Hypercube in point-interval queries which exactly match the key ordering (as described in section Compound Keys and Intervals (B-Trees)).

3.9.6 Hypercube Trees as Secondary Indexes

The Hypercube index technology also can be used for secondary indexes of a table. For example:

```
CREATE INDEX quot_qon_price ON quotations(qonorder,price,suppno,partno)
HCKEY IS qonorder,price
```

A 2 dimensional Hypercube index is created. This would be reasonable if queries with interval conditions on qonorder and price are frequent. In this example, it is expected that suppno and partno also are referenced very often (for example in the SELECT list). To avoid tuple materialization, the 2 fields suppno and partno have been arranged to be contained in the index but do not participate in the key.

As already explained, a Hypercube tree should have no fields as part of the key which typically are not searched for - therefore fields and keys can be specified separately in the Hypercube case.

It is possible to have several secondary indexes where some are of B-Tree type and others of Hypercube Tree type. They also can be used simultaneously in one query and intersected as already explained in subsection Multiple Secondary Indexes.

3.10 Join Queries

This chapter describes, how joins are optimized. To give a precise description a couple of terms have to be defined first. Then the join strategy is outlined and finally some methods are discussed that allow the user to change the join generation method.

3.10.1 Terms

3.10.1.1 Join (Equi-Join)

A join of two tables t and s is a query of the form

```

select ...
sfrom s, t
where s.att1 = t.att2
and P

```

The join is said to be performed over the fields att1 of s and att2 of t. P denotes a further predicate that applies to the participating tables (P of course might be missing, but in most cases additional restrictions are present). Clearly, joins can be nested, i.e. more than two tables can participate, but only two join tables are considered at the beginning of this chapter.

3.10.1.2 Local Restrictions

Most often, not all result tuples of a join are fetched, but additional predicates are specified whose components often refer to one of the two tables only.

Predicates of this kind are called *local restrictions*.

Example:

```

select ...
from s, t
where s.att1 = t.att2
and s.x = 17

```

The above example is a join over tables s and t where a local restriction on s is defined ($s.x = 17$).

Example:

```

select price
from suppliers s, quotations q
where s.suppno = q.suppno AND
      s.name = 'ATLANTIS CO.' AND
      q.partno = 232

```

```

(N0:sel { 1 "price" } --#tup: 1
  (N1:times { proj 1(3) } --#tup: 1
    (N2:restr --#tup: 1
      (N3:rel { "suppliers" proj 2(1 2) } --#tup: 7
        )
      (N4:eq { }

```

```

(N5:attr { N2[2] } )
(N6:const { 'ATLANTIS CO.' char(12) } )))
(N7:rel { "quotations" proj 1(3) } --#tup: 1
(N8:times { } --#tup: 1
(N9:ivmk { eq } --#tup: 1
(N10:attr { N1[1] } ))
(N11:ivmk { eq } --#tup: 1
(N12:const { 232 integer } ))))))))

```

The predicate of this join query exhibits two local restrictions, one on each of the joined tables.

The optimizer always creates access plans where local restrictions are processed before the tables are joined. This is valid in any of the join strategies described below. The operator tree shows the evaluation of the local restrictions for both relations suppliers and quotations. In the operator tree, the join is indicated by the `times` node. In this case, the join is a nested loop join.

3.10.1.3 Sort Merge Join

The Sort Merge Join works as follows:

Given the join pattern:

```

SELECT price
FROM s, t
WHERE s.att1 = t.att2 AND P

```

the following steps are performed:

- (SM1) sorting all tuples in `s` on `att1` (after local restrictions on `s` have been applied)
- (SM2) sorting all tuples in `t` on `att2` (after local restrictions on `t` have been applied)
- (SM3) merging tuples from sorted `s` and sorted `t` to build up the join partners.

The application of local restrictions *before* sorting and merging is not a characteristic of the Sort Merge Join but also applies to other join algorithms.

Of course, the sorting phase can be suppressed on either side if the table happens to be sorted on the join field. This depends on the keys, secondary indexes and local restrictions (if any) of the join partners.

```

(N0:sel { 1 "price" } --#tup: 1
  (N1:mjoin { 1=1 proj 1(4) } --#tup: 1
    (N2:proj --#tup: 1
      (N3:restr --#tup: 1
        (N4:rel { "suppliers" proj 2(1 2) } --#tup: 7
          )
          (N5:eq { }
            (N6:attr { N3[2] } )
            (N7:const { 'ATLANTIS CO.' char(12) } )))
          (N8:build
            (N9:attr { N2[1] } )))
        (N10:restr --#tup: 1
          (N11:rel { "quotations" proj 3(1 2 3) } --#tup: 16
            )
            (N12:eq { }
              (N13:attr { N10[2] } )
              (N14:const { 232 integer } )))))
    )
  )

```

Consider the example above. The operator tree with a sort merge join is shown here. The supplier table has a local restriction on name, but no secondary index on name.

The local restriction is therefore evaluated by a complete scan over the primary index and the tuples arrive sorted on the suppno which is the join field. No sorting must be applied on suppliers, instead the tuples can be merged "on-the-fly".

The table quotations also has a local restriction (on partno) but no secondary index on partno. Like for the table suppliers, the primary index is scanned and the tuples arrive sorted on field suppno.

Therefore, both sort operations can be avoided in this example. If, however, suppliers had a secondary index on name, then the optimizer would use it to evaluate the local restriction (name = 'ATLANTIS '). Perhaps, this would evaluate the restriction much faster than the primary index scan, but now the remaining tuples would have to be sorted on suppno for the join operation.

This shows that the existence of secondary indexes also influences the performance of joins. The following operator tree uses the secondary index quot_partno(partno) and can evaluate the local restriction as index access (and following tuple materialization on the base table quotations). In this case, the materialized tuples must be sorted, because the order of the tuples from the index do not correspond to the join predicate.

```

(N0:sel { 1 "price" } --#tup: 0
  (N1:mjoin { 1=1 proj 1(3) } --#tup: 0
    (N2:proj --#tup: 0

```

```

(N3:restr  --#tup: 0
  (N4:rel  { "suppliers"   proj 2(1 2) }  --#tup: 0
  )
  (N5:eq   { }
    (N6:attr { N3[2] } )
    (N7:const { 'ATLANTIS CO.' char(12) } )))
(N8:build
  (N9:attr { N2[1] } )))
(N10:sort { +1 , }  --#tup: 0
  (N11:proj  --#tup: 0
    (N12:mat { "quotations" }  --#tup: 0
      (N13:proj  --#tup: 0
        (N14:index { "quot_partno"   proj 2(2 2)
                    user_tablename="quotations" }  --#tup: 0
          (N15:ivmk { eq }  --#tup: 0
            (N16:const { 232 integer } )))
        (N17:build
          (N18:attr { N13[1] } )))))
    (N19:build
      (N20:attr { N11[1] } )
      (N21:attr { N11[3] } ))))))))

```

3.10.1.4 Nested Loop Join

Consider again the join

```

SELECT ...
FROM s, t
WHERE s.att1 = t.att2 AND P

```

In the sequel, the nested loop join method over s and t is described. It is abbreviated $NL(s,t)$. It works as follows:

- (NL1) fetch a tuple of s (which fulfills the local restrictions on s if there are any)
- (NL2) compute its partner tuple(s) in t (with application of local restrictions on t if there are any)
- (NL n) repeat steps (NL1) and (NL2) until no tuples in s remain.

The performance of this join algorithm mainly depends on 2 factors, namely the time to perform step NL2 and the number of repetitions of NL2.

The repetition count for NL2 depends on the selectivity of the local restriction on table *s* if there is any, otherwise on the number of tuples in table *s*. If the local restriction on *s* is highly selective (i.e. very few tuples remain) then the repetition count is small and there is a good chance that the algorithm works well.

The time to perform one incarnation of step NL2 depends on the access paths on table *t*. Performing step NL2 is similar to the query

```
SELECT ..
FROM t
WHERE t.att2 = x1
```

where the value of *x1* comes from *s.att1* and is fixed. Here it is assumed that no local restrictions on *t* were present in the original query, otherwise there would exist an additional predicate. The above query is executed according to the descriptions for single table queries in the first chapters. The main question is if there is a index (primary or secondary) on *t.att2*. If so, then step NL2 is fast else (for large *t*) it is presumably much too slow to be performed many times.

In summary, the NL(*s,t*) method works well if the input cardinality of table *s* (including local restrictions) is small and the join field of *t* is indexed.

It is easy to see that, unlike the Sort Merge Join, the roles of both input tables in NL(*s,t*) are not the same, i.e. NL(*s,t*) is not the same as NL(*t,s*). For a given query in the above scheme, NL(*s,t*) may work perfectly but NL(*t,s*) may perform very poor or vice versa.

With the Sort Merge Join, both sides behave symmetrical, so SM(*s,t*) and SM(*t,s*) perform equally and are not distinguished in the sequel.

The following query is the same as in the section before:

```
select price
from suppliers s, quotations q
where s.suppno = q.suppno AND
      s.name = 'ATLANTIS CO.' AND
      q.partno = 232
```

```
(N0:sel { 1 "price" } --#tup: 1
 (N1:times { proj 1(3) } --#tup: 1
  (N2:restr --#tup: 1
   (N3:rel { "suppliers" proj 2(1 2) } --#tup: 7
    )
   (N4:eq { }
    (N5:attr { N2[2] } )
    (N6:const { 'ATLANTIS CO.' char(12) } )))
```

```

(N7:rel { "quotations"  proj 1(3) } --#tup: 1
  (N8:times {  } --#tup: 1
    (N9:ivmk { eq  } --#tup: 1
      (N10:attr { N1[1] } ))
    (N11:ivmk { eq  } --#tup: 1
      (N12:const { 232 integer } )))))

```

The nested loop join is specified by the `times` node with two children nodes. The first child (left child) is a sub tree containing so far evaluated tuples. These tuples may originate directly from a relation (`rel` node) or from a restriction on a relation (`restr` node) or any sub operator tree. The second child (right child) must provide a direct access to a B-Tree, e.g., to the base relation via direct key access (and further restrictions may be evaluated, too).

In the example above the left son is the restriction on `suppliers` (`s.name = 'ATLANTIS CO.'`), i.e., node N2. The right son is the relation `quotations` with the restriction `q.partno = 232`, i.e., N7.

3.10.2 The Join Optimization Rule

Transbase[®] makes the following choice between the join methods:

- (J1) A nested loop join between `s` and `t` $NL(s,t)$ is generated if the input resulting from `s` is small and `t` is indexed on the join field.
- (J2) By definition, the input resulting from `s` is small if the query exhibits a local restriction on `s` or `s` is the result of a previous nested loop join.
- (J3) In all other cases, a sort merge join $SM(s,t)$ or $SM(t,s)$ is generated.

Rule (J2) says that for a 3 table join, if the result of a nested loop join between `s` and `t` is joined with `u` and an access path on the join partner in `u` exists, another nested loop join is done.

- (J4) If for a query, rules J1 and J2 would permit both $NL(s,t)$ and $NL(t,s)$, then the order of notation of the join predicate defines the order, i.e. a predicate `s.att1 = t.att2` implies $NL(s,t)$ whereas `t.att2 = s.att1` implies $NL(t,s)$. The order of the notation of the tables in the `FROM` clause has no effect on the join strategy.

3.10.2.1 Examples for Join Queries

Assume the sample database from chapter 3.3 without any secondary indexes.

Example:

```

SELECT ...
FROM suppliers s, quotations q
WHERE s.suppno = q.suppno AND
      s.name = 'XY Company' AND
      q.partno = 210

```

The optimizer generates a NL(s,q) because s has a local restriction (s.name = 'XY ...') and q.suppno is indexed.

If the join predicate were written as q.suppno = s.suppno, then NL(q,s) would be generated (Rule J4).

Example:

```

SELECT ...
FROM suppliers s, quotations q
WHERE s.suppno = q.suppno AND q.partno = 210

```

The optimizer generates a NL(q,s) because q has a local restriction q.partno = 210 and s.suppno is indexed.

Example:

```

SELECT *
FROM quotations q, inventory i
WHERE i.partno = q.partno AND
      i.description = 'BOLT'

```

```

(N0:sel { 8 "suppno" "partno" "price" "deliv_time" "qonorder"
         "partno" "description" "qonhand" } --#tup: 0
(N1:mjoin { 2=1 } --#tup: 0
(N2:sort { +2 , } --#tup: 1
(N3:rel { "quotations" proj 5(1 2 3 4 5) } --#tup: 16
))
(N4:restr --#tup: 0
(N5:rel { "inventory" proj 3(1 2 3) } --#tup: 9
)
(N6:eq { }
(N7:attr { N4[2] } )
(N8:const { 'BOLT*' char(5) } ))))

```

The optimizer generates a SM join because neither NL(q,i) nor NL(i,q) is permitted by J1 and J2.

If there were a secondary index on q.partno then a NL(i,q) would be applied.

```
(N0:sel {      8 "suppno" "partno" "price" "deliv_time" "qonorder"
              "partno" "description" "qonhand" } --#tup: 0
(N1:times {   proj 8(4 5 6 7 8 1 2 3)   } --#tup: 0
(N2:restr  --#tup: 0
(N3:rel   { "inventory"   proj 3(1 2 3) } --#tup: 9
)
(N4:eq   { }
(N5:attr { N2[2] } )
(N6:const { 'BOLT*'   char(5) } )))
(N7:mat  { "quotations" } --#tup: 0
(N8:index { "quot_partno"   proj 1(2)
            user_tablename="quotations" } --#tup: 0
(N9:ivmk  { eq   } --#tup: 0
(N10:attr { N1[1] } ))))
```

3.10.3 Multi-Way Joins

For joins where more than 2 tables are involved the notion of a join tree must be introduced and explained.

Example:

```
SELECT price
FROM suppliers s, quotations q, inventory i
WHERE s.suppno = q.suppno AND
      q.partno = i.partno AND
      s.name = ''ATLANTIS CO.''
```

The following join orders are possible: either s and q are joined and the join result is joined with i, or s is joined with the join result of q and i.

The two alternative join orders are depicted in figure 3.1.

The JOIN node either stands for the SM(.) or NL(.) join method.

For example 3.10.3 above, Transbase[®] would generate the join NL(NL(s,q),i). Figure 3.2 shows the corresponding join graph. The operator tree for this join graph is shown in the following:

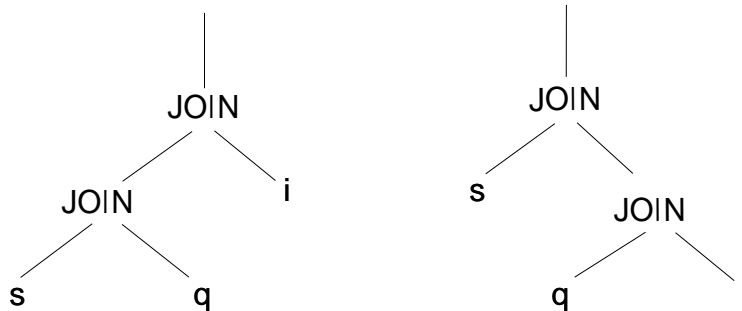


Figure 3.1: Two alternative join orders

```

(N0:sel { 1 "price" } --#tup: 3
  (N1:times { proj 1(2) } --#tup: 3
    (N2:times { proj 2(3 4) } --#tup: 3
      (N3:restr --#tup: 1
        (N4:rel { "suppliers" proj 2(1 2) } --#tup: 7
          )
          (N5:eq { }
            (N6:attr { N3[2] } )
            (N7:const { 'ATLANTIS CO.' char(12) } )))
          (N8:rel { "quotations" proj 2(2 3) } --#tup: 3
            (N9:ivmk { eq } --#tup: 1
              (N10:attr { N2[1] } )))))
          (N11:rel { "inventory" proj 0() } --#tup: 3
            (N12:ivmk { eq } --#tup: 1
              (N13:attr { N1[1] } ))))))
  )
)

```

Note that $NL(s,q)$ is chosen according to $J1$ and $J2$ and then another $NL(result,i)$ is chosen because according to $J2$ the result of the first join is again suited as left operand for a NL join.

It has to be noted that in this example the whole execution plan perfectly runs "on-the-fly" - this is a property of chained nested loop joins.

If table `inventory` were not indexed on `partno` (in the sample database schema, `partno` is the primary key) then a tree as shown in figure 3.3 would be generated.

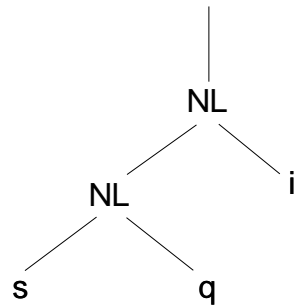
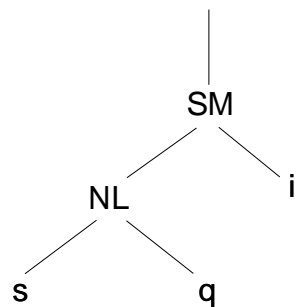
Figure 3.2: Join graph for $NL(NL(s,q),i)$ 

Figure 3.3: Operator tree without index

The result of $NL(s,q)$ would be "sort-merged" with table *i*. Both inputs would have to be sorted on *partno* before the merge can start.

Example:

```
SELECT price
FROM suppliers s, quotations q, inventory i
WHERE s.suppno = q.suppno AND
      q.partno = i.partno AND
      q.price > 1000
```

With the keys as defined in the sample database, the access plan would be changed as depicted in figure 3.4.

Achtung!!!! Der stimmt so nicht (Reihenfolge der Joins!!)

```
(N0:sel { 1 "PRICE" } --#tup: 0
  (N1:times { proj 1(3) } --#tup: 0
    (N2:times { proj 3(1 2 3) } --#tup: 0
      (N3:restr --#tup: 0
        (N4:rel { "quotations" proj 3(1 2 3) } --#tup: 16
          )
          (N5:gt { }
            (N6:attr { N3[3] } )
            (N7:const { 1000 integer } )))
          (N8:rel { "suppliers" proj 0() } --#tup: 0
            (N9:ivmk { eq } --#tup: 0
              (N10:attr { N2[1] } ))))
          (N11:rel { "inventory" proj 0() } --#tup: 0
            (N12:ivmk { eq } --#tup: 0
              (N13:attr { N1[2] } ))))))
```

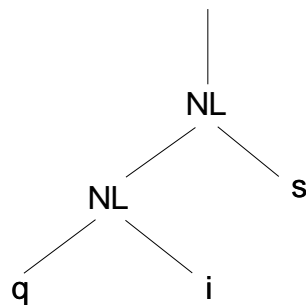


Figure 3.4: Operator Tree

If suppliers were not indexed on `suppno`, we would have 2 plans (considered equivalent) as shown in figure 3.5.

Both trees are considered equivalent because SM is symmetrical.

The following rule applies for multi-way joins:

(J5) In join queries, the optimizer arranges as many NL joins as are permitted by rules (J1) to (J4). In this phase the join tree is built in the order as the join predicates are written. All remaining join predicates are then handled via SM joins.

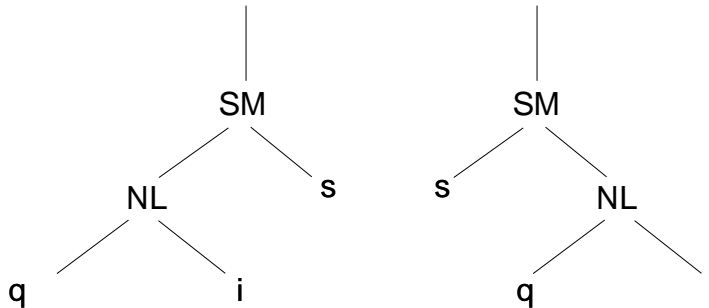


Figure 3.5: Equivalent plans

3.10.4 User Control of Transbase Join Strategy

This chapter describes how the user can influence the Transbase[®] join strategy. For this, 2 methods are provided, namely an explicit query construct and a global optimizer switch.

3.10.4.1 Keywords NLJOIN and SMJOIN

The user can override the Transbase[®] join strategy by reformulating the join predicate using the keywords NLJOIN and SMJOIN. Both keywords stand for the equality operator =. Only the join order is influenced but the query results of course remain the same.

In the following examples, it is explained what joins are generated and what joins would be generated if = were written instead.

Example: adapted from example 3.10.2.1

```

SELECT ...
FROM suppliers s, quotations q
WHERE s.suppno SMJOIN q.suppno AND
      q.partno = 210
  
```

Instead of the NL(q,s) join in example 3.10.2.1, a SM join would be generated here. Both sorting operations would be suppressed here because suppliers arrive already sorted on suppno as well as quotations.

Example:

```
select ...
from quotations q, inventory i
where q.partno NLJOIN i.partno AND
      i.description = 'BOLT'
```

A NL(q,i) join would be generated here.

Example: adapted from example 3.10.2.1

```
SELECT ...
FROM quotations q, inventory i
WHERE i.partno NLJOIN q.partno AND
      i.description = 'BOLT'
```

Instead of a SM join, a NL(i,q) join would be generated here.

The last two examples show that the NLJOIN formulation respects the order in which the join predicate is written. Note that the last example probably does not perform well because quotations is not indexed on partno. This is further discussed in chapter (Risks of Explicitly Specified Join Methods).

Also note that a NLJOIN or SMJOIN formulation for predicates other than join predicates is not an error but just has the meaning of = (e.g. q.price NLJOIN 1000 is the same as q.price = 1000).

3.10.4.2 NLJOIN and SMJOIN in Multi-Way Join Queries

In multi-way join queries, the NLJOIN and SMJOIN mechanism provide the possibility to specify in detail how the join tree is built. For this, the NLJOIN and SMJOIN formulation can be freely mixed. The optimizer takes all NLJOIN and SMJOIN predicates and builds the join tree in the order in which the NLJOIN and SMJOIN predicates are written. Finally all remaining join predicates specified with = (if there remain any) are taken and the join tree is completed. In this second phase, the join strategies as described by rules (J1) to (J5) are used.

Example:

```
SELECT ...
FROM q, r, s, t
WHERE r.f1 NLJOIN q.f1 AND
      s.f3 NLJOIN t.f3 AND
      r.f2 SMJOIN s.f2 AND
      < local restrictions >
```

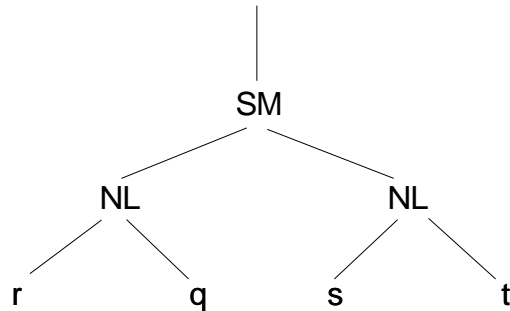


Figure 3.6: Join tree

The join tree built is shown in figure 3.6.

A rearrangement of the join predicate like below would lead to quite another join tree shown in figure 3.7.

Example:

```

SELECT ...
FROM q, r, s, t
WHERE r.f1 NLJOIN q.f1 AND
      r.f2 SMJOIN s.f2 AND
      s.f3 NLJOIN t.f3 AND
      < local restrictions >
  
```

3.10.4.3 Setting the Joinmode with Tbmode

Control of join algorithms can also be achieved by setting the optimizer in a special mode with the **TBMODE** statement:

TBMODE OPTIMIZER NLJOIN sets the optimizer in the **NLJOIN** mode, i.e. for join predicates formulated by = only **NL** joins are generated.

TBMODE OPTIMIZER SMJOIN sets the optimizer in the **SMJOIN** mode, i.e. for join predicates formulated by = only **SM** joins are generated.

TBMODE OPTIMIZER DEFAULT sets the optimizer back in the Transbase[®] default mode, i.e. it works as described by rules (J1) to (J5).

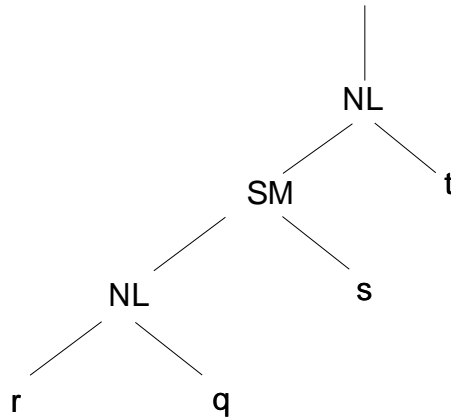


Figure 3.7: Alternative join tree

The explicit join control operands `NLJOIN` and `SMJOIN` in the queries override the `TBMODE` statement. Only `=` predicates are affected by the `TBMODE`. The `TBMODE` method is not as flexible as the explicit join control operands, but it has the advantage that the query formulation remains in accordance to ISO-SQL.

3.10.4.4 Risks of Explicitly Specified Join Methods

It has to be noted that user specified join strategies may perform almost arbitrarily poor if the user has specified a *bad* join strategy. Some examples:

Example:

```

SELECT ...
FROM quotations q, inventory i
WHERE i.partno NLJOIN q.partno AND
      i.description = 'BOLT'
  
```

Remember the `NL` algorithm from Chapter "Nested Loop Join". Table `q` has its highest key on `suppno`. If `q` has no secondary index on `partno` then for each tuple `i` (which fulfills the local restriction on `i.description`) the join partners in `q` must be searched via a total sequential scan on `q`. If the local restriction is not very selective and `quotations` is large then the query performs very poor.

Example:

```

SELECT ...
  
```

```

FROM suppliers s, quotations q, inventory i
WHERE q.partno SMJOIN i.partno AND
      s.supplyno NLJOIN q.supplyno AND
      < some local restrictions >

```

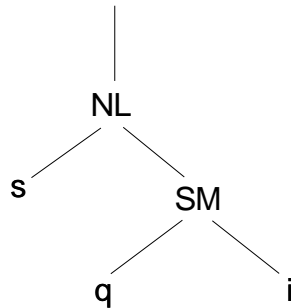


Figure 3.8: Execution plan

Independent of local restrictions or keys or indexes, this query is likely to perform extremely poor for large tables. The point is that the right operand of the NL join is itself a join. The right operand of NL, however, should be computable efficiently because it is **recomputed for each left input tuple** (its actual result depends on the actual left input tuple). The corresponding execution plan is shown in figure 3.8.

3.10.4.5 Hints for Join Query Optimization

The following characteristics of the join methods should help to build optimal join trees:

The right operand of NL must be very efficiently computable. The cardinality of the left operand of NL must be small. The sort orders of the NL input tuples play no role for the efficiency of NL.

The SM method sorts the input tuples on the join fields. If one or both sides arrive already sorted then the sorting is suppressed. Left and right side of SM input tuples may be interchanged without affecting performance.

The NL method produces tuples whose sort order is that of its left operand. The SM method produces tuples which are sorted on the join fields.

The sort order of base table tuples (i.e. tuples which are not yet joined) depends on the index (primary or secondary) which is used. This is described in the chapter about Single Table Queries.

3.10.4.6 Outer Joins

Outer Joins are handled with essentially the same algorithms than inner joins.

Especially, a LEFT OUTER Join between R and S is well done by a Nested Loop Join NL(R,S) or a Sort Merge Join.

A RIGHT OUTER Join between R and S is well done by a Nested Loop Join NL(S,R) or a Sort Merge Join.

A FULL OUTER Join is preferably done by a Sort Merge Join.

Also, explicit control mechanisms as described in 3.10.4 are applicable in the same way. The NLJOIN or SMJOIN clauses can be applied to the join predicate in the ON clause.

3.10.5 Aspects of Multi-Column Joins

A multi-column join between 2 tables is a join where more than one field pair is specified.

Example:

```
SELECT ...
FROM   s, t
WHERE  s.att1 = t.att2 AND
       s.att3 = t.att4 AND
       s.att5 = x1
```

Obviously, both conditions are treated with the same join strategy, i.e. it makes no sense to try a NL for the first condition and a SM for the second. When the Transbase[®] optimizer searches for a NL possibility it inspects all conditions independently. For example, if t were indexed on att4 (note also the local restriction on s), then Transbase[®] would apply a NL(s,t).

The only semantical unsoundness appears if a user specified join order explicitly by NLJOIN or SMJOIN and there is an inconsistency between multi columns, for example:

```
SELECT ...
FROM   s, t
WHERE  s.att1 SMJOIN t.att2 AND
       s.att3 NLJOIN t.att4 AND
       ....
```

As discussed, Transbase[®] builds the join tree in the order of the join specification, i.e. starts with SM(s,t) and then would simply integrate the second join condition into the SM join even its specification contradicts the first one. In other words, Transbase[®] ignores explicit join specifications if they refer to tree parts which have already been built but integrates them in a consistent manner.

3.11 Subqueries Delivering a Value

In TB-SQL, subqueries which deliver a value can appear in the **WHERE** clause or in the **SELECT** clause. They can be formulated wherever a value expression (field, constant, host variable) can be used. Their result is used like a value.

Example:

```
SELECT *
FROM quotations
WHERE suppno = (SELECT suppno
                 FROM suppliers
                 WHERE name = 'ATLANTIS CO.')
```

```
(N0:sel { updatable 5 "suppno" "partno" "price" "deliv_time"
          "qonorder" } --#tup: 3
  (N1:rel { "quotations" proj 5(1 2 3 4 5) } --#tup: 3
    (N2:ivmk { eq } --#tup: 1
      (N3:val
        (N4:proj --#tup: 1
          (N5:restr --#tup: 1
            (N6:rel { "suppliers" proj 2(1 2) } --#tup: 7
              )
            (N7:eq { }
              (N8:attr { N5[2] } )
              (N9:const { 'ATLANTIS CO.' char(12) } )))
          (N10:build
            (N11:attr { N4[1] } ))))))))
```

Example:

```
SELECT suppno, (SELECT description
                 FROM inventory i
                 WHERE i.partno = q.partno )
FROM quotations q
WHERE qonorder <> 0
```

```

(N0:sel { 2 "suppno" "column_1" } --#tup: 9
  (N1:proj --#tup: 9
    (N2:restr --#tup: 9
      (N3:rel { "quotations" proj 3(1 2 5) } --#tup: 16
        )
      (N4:ne { }
        (N5:attr { N2[3] } )
        (N6:const { 0 integer } )))
    (N7:build
      (N8:attr { N1[1] } )
      (N9:corr
        (N10:val
          (N11:rel { "inventory" proj 1(2) } --#tup: 9
            (N12:ivmk { eq } --#tup: 1
              (N13:attr { N1[2] } ) )))
          (N14:attr { N1[2] } )))))
  )
)

```

The first example shows a subquery which is uncorrelated, i.e. it delivers the same value for each of the tuples in quotations. Subqueries are uncorrelated if they do not contain a reference to the enclosing query blocks. As it is shown in the operator tree there is an equity predicate (`ivmk { eq }`) followed by the `val` node with the sub query as sub tree.

The second example has a subquery which is correlated (it has an outer reference to quotations `q`, namely `q.partno`). For this purpose, a special node `corr` is used (see description below).

Uncorrelated value subqueries are optimized on their own as discussed so far in the preceding chapters. At runtime, they are evaluated once and their result is reused as often as it is needed. The surrounding query block is optimized as if the enclosed subquery were a value.

Therefore, example 3.11 is optimized as follows: one sequential scan on suppliers (unless there is a secondary index on 'name', allowing a direct access), the resulting `suppno` value is used as argument for a key access on quotations.

Correlated value subqueries are optimized at their own by treating the outer reference like a constant value. At runtime they are reevaluated as often as the value of the outer reference changes (at runtime, the last result value and the last value of the outer reference are remembered).

Therefore, example 3.11 is optimized as follows:

One sequential scan on quotations (unless there is a secondary index on `qonorder`); for each result tuple of quotations it is tested whether the value `q.partno` is the same as in the last result tuple; if not then the subquery on inventory is reevaluated which consists in a key access on its primary index with `q.partno` as key value.

Note as a side remark that the last example works very similar to a NL(q,i) join query. The difference is that a NL join evaluation plan does not remember the result of the last call of its right operand because in general this is no value but an arbitrarily large tuple set.

3.12 Subqueries with IN and EXISTS

Subqueries in the WHERE clause which are connected by the operator IN or EXISTS are optimized in a special way. Semantically they form a kind of implicit join queries.

Example:

```
SELECT s.name
FROM   suppliers s
WHERE  s.suppno IN (SELECT q.suppno
                   FROM   quotations q
                   where qonorder <> 0)
```

Example:

```
SELECT s.name
FROM   suppliers s
WHERE  EXISTS (SELECT *
              FROM   quotations q
              WHERE  s.suppno = q.suppno AND
              qonorder <> 0)
```

Both query schemata deliver the same result. They also are optimized in the same way. The optimizer treats the queries similar (but not quite identical) to the following join query, namely:

Example:

```
SELECT s.name
FROM   suppliers s, quotations q
WHERE  s.suppno = q.suppno AND
       qonorder <> 0)
```

In fact, the query form of example 3.12 would not deliver the same result: If a supplier appears more than once in the (restricted) quotations table then the query result of example 3.12 would produce duplicates in contrast to examples 3.12 and 3.12.

Adding a `DISTINCT` clause in the `SELECT` clause of example 3.12 is not a remedy because this now would eliminate possible duplicates of supplier names which appear in examples 3.12 and 3.12.

Therefore, the access plan generated from examples 3.12 and 3.12. eliminates duplicates on the quotations side of the join only.

The optimization of examples 3.12 and 3.12 now is similar to that of example 3.12 but with the difference that never a `NL(s,q)` schema is generated but only `NL(q,s)` and `SM(q,s)` is possible. The generation of `NL(s,q)` is prohibited by the elimination of duplicates on the `q` side - these are technical reasons which could be eliminated in future versions.

The choice between `SM` and `NL` is done like in "normal" join queries. In the example above, a `NL(q,s)` join would be made because quotations has a local restriction and suppliers is indexed on the join attribute `suppno`.

It is to be noted that the forms 3.12 and 3.12 cannot be influenced by explicit join control mechanisms like `NLJOIN` or `SMJOIN`.

3.13 Evaluation of UNION, INTERSECT, DIFF

The operators `UNION`, `INTERSECT`, `DIFF` combine query blocks delivering tuples of the same arity and types. They all are evaluated by sorting the input tuples on both sides on all fields (in ascending order and ascending weights on position) and then by a merge algorithm. If one or both sides happen to be already sorted then the optimizer recognizes that the sort operation(s) can be suppressed. The optimization and complexity of the `UNION`, `INTERSECT` and `DIFF` operator therefore are quite similar to that of a `SM` join.

Example:

```
SELECT suppno
FROM suppliers
DIFF
SELECT suppno
FROM quotations
```

The above query delivers the supplier numbers of those suppliers which do not appear in quotations. In this example the input tuples on both sides appear sorted on `suppno` so the sort operations can be suppressed and the query runs as a linear merge scan.

Example:

```
SELECT partno
FROM   inventory
INTERSECT
SELECT partno
FROM   quotations
```

This example delivers the part numbers of inventory which appear in quotations. Here the input tuples from quotations projected on partno have to be sorted for the INTERSECT operation. (unless there is a secondary index with highest key on partno).

The UNION operator works analogously. For the UNION operator it has to be noted that the sorting operations are only necessary to realize the duplicate elimination which is required by ISO-SQL. In contrast to that the UNION ALL operator does not eliminate duplicates and thus does not sort its input and is very cheap at runtime.

For the UNION operator also note chapter (GREEDY Mode for DISTINCT and UNION).

3.14 Queries with NOT IN and NOT EXISTS

Queries with the NOT IN or NOT EXISTS operator often cause performance problems. Consider the following

Example:

```
SELECT *
FROM   suppliers
WHERE  suppno NOT IN
      (SELECT suppno FROM quotations)
```

Example:

```
SELECT *
FROM   suppliers s
WHERE  NOT EXISTS
      (SELECT *
       FROM   quotations q
       WHERE  q.suppno = s.suppno)
```

Except to a very subtle difference with respect to NULL values the 2 queries are equivalent. In fact the optimizer transforms query patterns shown in example 3.14 into equivalent ones shown in example 3.14 and then optimizes the subquery with respect to key access (use s.suppno for key access on quotations).

This query evaluation resembles very much a NL(s,q) strategy and therefore also runs into the NL performance problems which arise if the "left" input cardinality (suppliers) is not small or - even worse - if the "right" operand cannot be evaluated efficiently. Consider the case if quotations were not indexed on suppno: a total sequential scan of quotations would be necessary for each supplier tuple!

In such cases it would be much better to merge suppliers and quotations on suppno (possibly after sorting them which however is not necessary in this example) and find the suppliers tuples on the fly on the merge. This is what the DIFF operator does.

Example:

```
SELECT suppno
FROM   suppliers
DIFF
SELECT suppno
FROM   quotations
```

Note, however, that it is up to the user to reformulate the query to a DIFF query if the NOT IN or NOT EXISTS variant runs into trouble. Also note that the DIFF variant does not always run better because it sometimes has to sort its input, but it tends to outperform the NOT IN/NOT EXISTS if the left operand is large.

3.15 Queries with GROUP BY and DISTINCT

The GROUP BY clause is evaluated by sorting the input tuples (coming from the WHERE clause if there is one) on the specified GROUP attributes. As always, the sort operation is suppressed if the tuples arrive already sorted.

Example:

```
SELECT sum ( price * qonorder )
FROM   quotations
GROUP BY partno
```

```

(N0:sel { 1 "column_0" } --#tup: 9
  (N1:proj --#tup: 9
    (N2:group { [ 1 ] sum[2] } --#tup: 9
      (N3:sort { +1 , } --#tup: 16
        (N4:proj --#tup: 16
          (N5:rel { "quotations" proj 3(2 3 5) } --#tup: 16
            )
          (N6:build
            (N7:attr { N4[1] } )
            (N8:mul
              (N9:attr { N4[2] } )
              (N10:cast { numeric(10,0) }
                (N11:attr { N4[3] } ))))))))
      (N12:build
        (N13:attr { N1[2] } )))
  )
)

```

Here the quotations tuples are sorted on partno, then the **GROUP** operation is done with a linear scan (one result tuple can be produced whenever the partno value changes). If quotations had a secondary index on partno then it is not used unless the index also contained the 2 other needed fields (price, qonorder). The rule is that a secondary index is not used to fulfill sorting purposes if an additional tuple materialization were necessary.

Queries with **DISTINCT** in the **SELECT** clause are treated similarly. The result tuples are constructed according to the **SELECT** clause, then are sorted on all result fields (unless they are already sorted) and then are filtered to eliminate duplicates.

Example:

```

SELECT DISTINCT name
FROM suppliers

```

```

(N0:sel { 1 "name" } --#tup: 7
  (N1:uniq { } --#tup: 7
    (N2:sort { +1 , } --#tup: 7
      (N3:rel { "suppliers" proj 1(2) } --#tup: 7
        )))
  )
)

```

If suppliers had a secondary index on name, it would be used to save the required sorting on name otherwise the base tuples of suppliers would be projected on name, then sorted, then filtered.

Queries with the **DISTINCT** clause inside a set function are treated differently:

Example:

```
SELECT COUNT(*), COUNT(DISTINCT name), COUNT(DISTINCT address)
FROM suppliers
```

```
(N0:sel { 3 "column_0" "column_1" "column_2" } --#tup: 1
  (N1:proj --#tup: 1
    (N2:group { count[distinct 1] count[distinct 2] count[*] }
      --#tup: 1
      (N3:rel { "suppliers" proj 2(3 2) } --#tup: 7
        ))
    (N4:build
      (N5:attr { N1[3] } )
      (N6:attr { N1[2] } )
      (N7:attr { N1[1] } ))))
```

The evaluation technique here is not sorting because the required sort orders can contradict as in the above example. Instead, all occurring values of name and address are inserted on the fly into appropriate data structures to recognize duplicates.

3.16 INSERT, UPDATE, DELETE Queries

3.16.1 INSERT Queries

INSERT queries with the VALUES clause show no optimization potential, except for subqueries in the VALUES clause. Like any other subqueries, they are optimized at their own as discussed.

INSERT queries specified with a SELECT query are optimized such that the tuples resulting from the SELECT block are sorted on the primary key positions of the target table. Thus it is assured that each page of the target B-Tree must be fetched at most once. The sorting is suppressed if the tuples to be inserted arrive already sorted from the SELECT clause.

It has to be noted that INSERT . . . SELECT queries may be slow if one or more secondary indexes exist and the number of tuples to be inserted is very large. The reason is that the secondary index tuples are not sorted on the (secondary) keys before insertion but are inserted on-the-fly. In extreme cases it may be faster to drop the secondary index(es) before the INSERT query and to rebuild them after the INSERT query because the CREATE INDEX algorithm works with sorting before insertion.

3.16.2 DELETE Queries

DELETE queries without a WHERE clause (i.e. deletion of all tuples in a table) presently are not optimized and thus run slowly. It is preferable to replace this query by a sequence of DROP TABLE .. and CREATE TABLE .. which runs very fast and with very few recovery space overhead.

DELETE queries with a WHERE clause are optimized like SELECT queries with an analogous WHERE clause. This means that key accesses on primary or secondary indexes as well as the discussed optimizations in IN or EXISTS predicates are performed.

3.16.3 UPDATE Queries

Similar to the DELETE queries, UPDATE queries which have a WHERE clause are optimized like analogous SELECT queries as far as the WHERE clause is concerned. For the maintenance of secondary indexes the optimizer takes care that only updated fields give rise to index maintenance.

Additionally, the optimizer checks whether any key value of the primary index is updated. If so, the update procedure must be run in 2 phases - namely a deletion phase followed by a insertion phase of the updated tuples. Consider for example an update query which adds the value 1000 to each suppnno in suppliers. A single pass update by a sequential scan would move each supplier tuple forward and the sequential scan would see the updated tuple once more and would need a update flag to recognize that the tuple has already been updated. To run these (seldom) cases correctly a 2 phase procedure is done.

The effect is that updates of key values run more slowly than updates of non-key values only.

3.16.4 Modification Queries with Self Reference

In rare but sometimes very important cases, the INSERT, UPDATE or DELETE query has subqueries with references to the target table. This is called a self referencing modification queries and is allowed in Transbase[®] SQL (but not in ISO-SQL).

Example:

```
UPDATE quotations q1
SET   price = price * 0.9
WHERE price = (SELECT max (price)
              FROM quotations q2
              WHERE q2.partno = q1.partno)
```

All self-referencing modification queries are run in a 2 phase procedure to avoid the influence of updated values on the retrieval semantics. The (unavoidable) effect is that self-referencing modification queries typically run more slowly than non self-referencing queries.

3.17 The Data Spooler

The statements for spooling data from tables or from queries into files

(`SPOOL INTO <file> SELECT ..`) are optimized like the `SELECT` queries that they contain.

The `SPOOL` statement for spooling data from file into a table

(`SPOOL <table> FROM <file>`) deserves some additional discussion. Let us first assume that no secondary index exists on the target table. If the records in the file are sorted on the primary key positions of the target table then the performance in general is quite high. If, additionally, the target table is empty then maximal performance is achieved.

On the other hand, if the records are not sorted on the primary keys then the spool procedure first inserts all tuples (records) which arrive in sort order, i.e. all tuples which contradict the sort order (i.e. which are not greater than the last inserted one) are temporarily stored in a auxiliary file. In the second phase the tuples in the auxiliary file are sorted and, in the third phase, are inserted in the target table.

If few tuples are out of sort order this procedure is almost as fast as if all were sorted. If very many tuples are out of sort order this procedure is by far superior to insertion on-the-fly because the latter would possibly fetch the data pages more than once. The worst case would cause several I/Os for each tuple.

However, sorting requires some additional disk space. For extreme large unsorted spoolfiles, it may be advantageous to split the input into several pieces and to spool them piecewise.

If secondary indexes exist, then 2 cases have to be distinguished. The advantageous case is that the target table is empty. In this case the secondary indexes are built after all tuples have been spooled into the primary index of the table. In other words, if the target table is empty then it makes no difference whether a `SPOOL` statement is followed by `CREATE INDEX` statements or the secondary indexes already exist when the `SPOOL` statement is processed.

If the target table is not empty then secondary indexes slow down the `SPOOL` statement. The reason is that the secondary index tuples are inserted on-the-fly and not with a store and sort and insert procedure. This is analogous to the `INSERT` statement. If large spool files have to be processed on non-empty tables,

it may be advantageous to drop the secondary indexes first and to recreate them after the spool procedure is finished.

3.18 GREEDY Mode for DISTINCT and UNION

Optimization of relational database queries typically tries to minimize the overall time to process the whole query. Very often, an optimization technique which successfully reduces the overall processing time has the side effect that the first result tuple is produced later than without the optimization. This means that "response" time (for first result tuple) and overall time often conflict.

A typical example is the SM join and NL join technique. Sometimes a SM join produces the best overall processing time but (if sorting is involved) needs a long time to produce the first result tuple. NL join, in contrast, typically produces the first result tuple very quickly.

To a (presently) very limited extent, the Transbase[®] optimizer offers a switch to choose between a **GREEDY** mode (quick response time for first tuple) and **GLOBAL** mode (small overall processing time). The optimizer always starts with **GLOBAL** mode. The statements to switch are:

```
TBMODE OPTIMIZER GREEDY
```

```
TBMODE OPTIMIZER GLOBAL
```

Presently, the **GREEDY** mode only influences the **DISTINCT** clause and queries with **UNION** operator. In **GLOBAL** mode, the **DISTINCT** operation is done by sorting followed by linear duplicate elimination. In **GREEDY** mode, each delivered tuple is additionally stored in a B-Tree and for each input tuple it is tested whether an identical tuple has already been delivered by searching it in the B-Tree. The overall time is longer because of the random accesses in the B-Tree, but the first tuples come very fast.

Likewise, for the **UNION** operator, the **GLOBAL** mode in general sorts the input tuples to make duplicate elimination whereas the **GREEDY** mode drives a **UNION ALL** followed by a **GREEDY DISTINCT** as described above.

It has to be noted here, that also the size of the tuple result buffer of the `tbkernel` process has an influence on the response time and overall time. The result buffer size can also be influenced by the `TBMODE` statement (see SQL Manual). A small buffer size favours **GREEDY**ness, a large buffer size reduces overall processing time.

3.19 Appendix

The sample database used in most of the examples:

suppliers		
suppno	name	address
51	DEFECTO PARTS	16 BUM ST., BROKEN HAND WY
52	VEUVIUS, INC.	512 ANCIENT BLVD., POMPEII NY
53	ATLANTIS CO.	8 OCEAN AVE., WASHINGTON DC
54	TITANIC PARTS	32 LARGE ST., BIG TOWN TX
57	EAGLE HARDWARE	64 TRANQUILITY PLACE, APOLLO MN
61	SKY PARTS	128 ORBIT BLVD., SIDNEY
64	KNIGHT LTD.	256 ARTHUR COURT, CAMELOT

inventory		
partno	description	qonhand
207	GEAR	75
209	CAM	50
221	BOLT	650
222	BOLT	1250
231	NUT	700
232	NUT	1100
241	WASHER	6000
285	WHEEL	350
295	BELT	85

quotations				
suppno	partno	price	delivery_time	qonorder
51	221	.30	10	50
51	231	0.10	10	0
53	222	0.25	15	0
53	232	0.10	15	200
53	241	0.08	15	0
54	209	18.00	21	0
54	221	0.10	30	150
54	231	0.04	30	200
54	241	0.02	30	200
57	285	21.00	4	0
57	295	8.50	21	24
61	221	0.20	21	0
61	222	0.20	21	200
61	241	0.05	21	0
64	207	29.00	14	20
64	209	19.50	7	7

The schema is defined by the following statements:

```
CREATE TABLE suppliers (  
    suppno    INTEGER,  
    name      CHAR(*),  
    address   CHAR(*))  
KEY IS suppno;  
  
CREATE TABLE inventory (  
    partno    INTEGER,  
    description CHAR(*),  
    qonhand   INTEGER)  
KEY IS partno;  
  
CREATE TABLE quotations (  
    suppno    INTEGER,  
    partno    INTEGER,  
    price     NUMERIC(8,2),  
    deliv_time INTEGER,  
    qonorder  INTEGER)  
KEY IS suppno, partno;
```