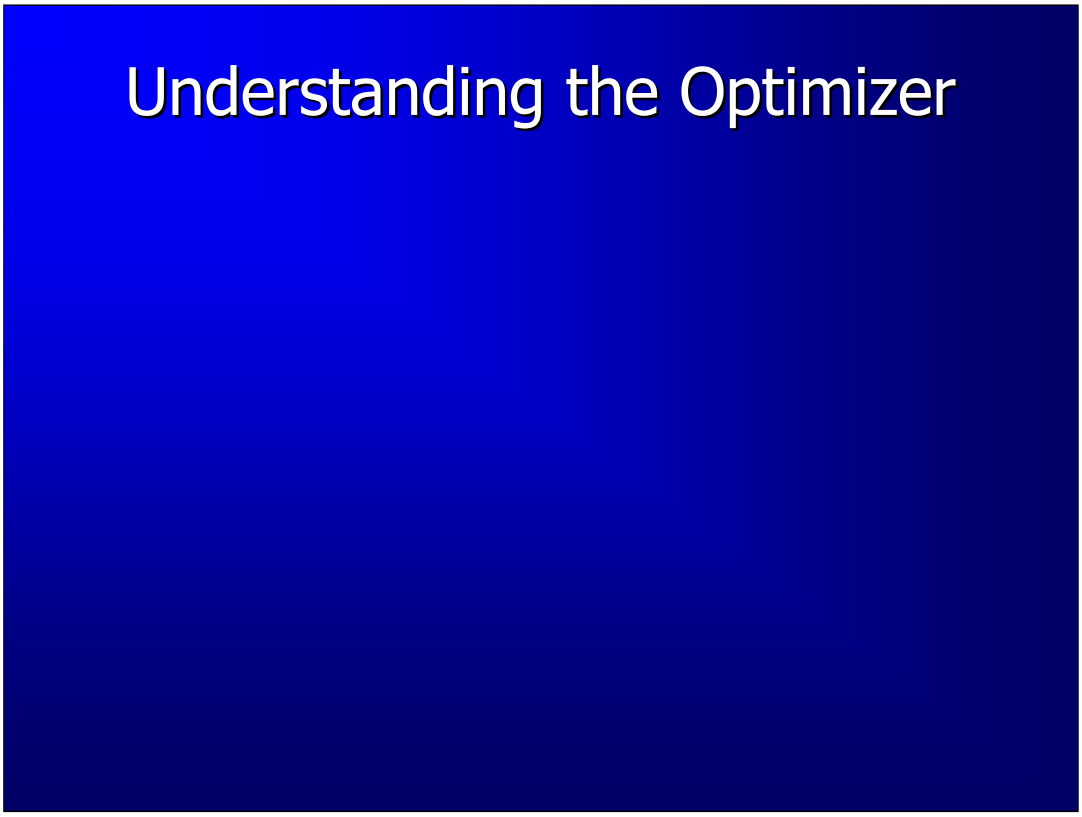
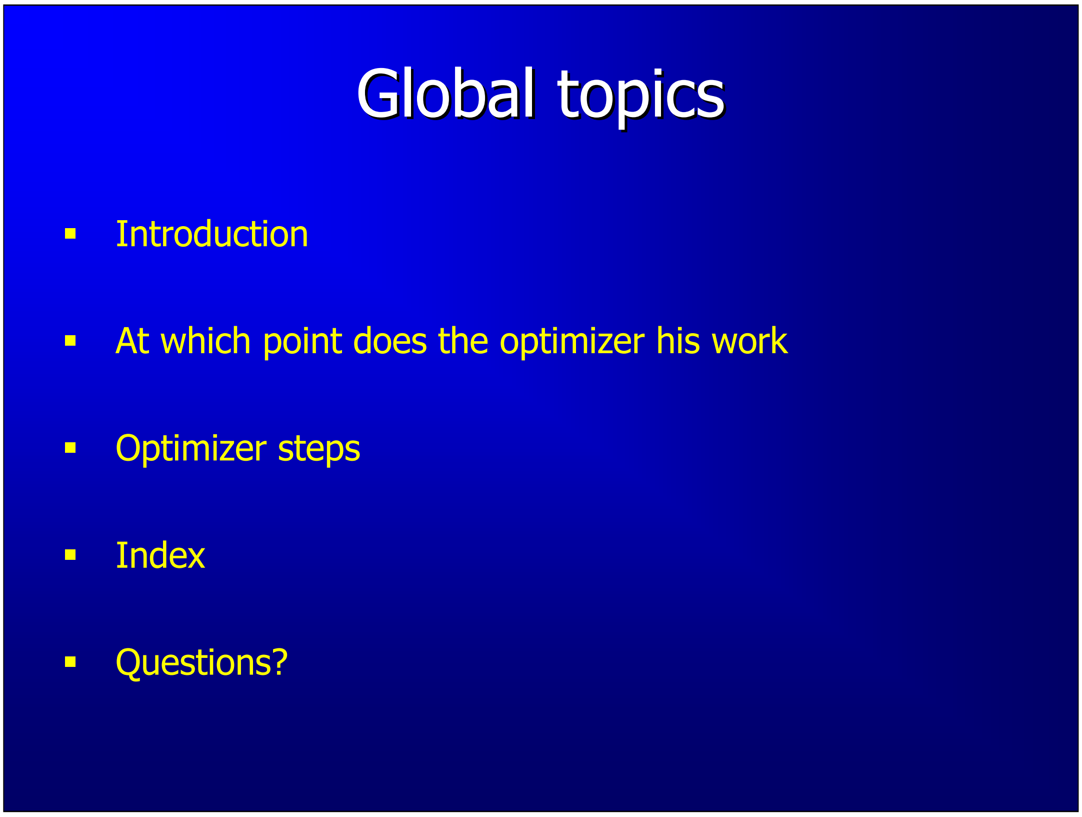


Understanding the Optimizer



Global topics

- Introduction
- At which point does the optimizer his work
- Optimizer steps
- Index
- Questions?

Introduction

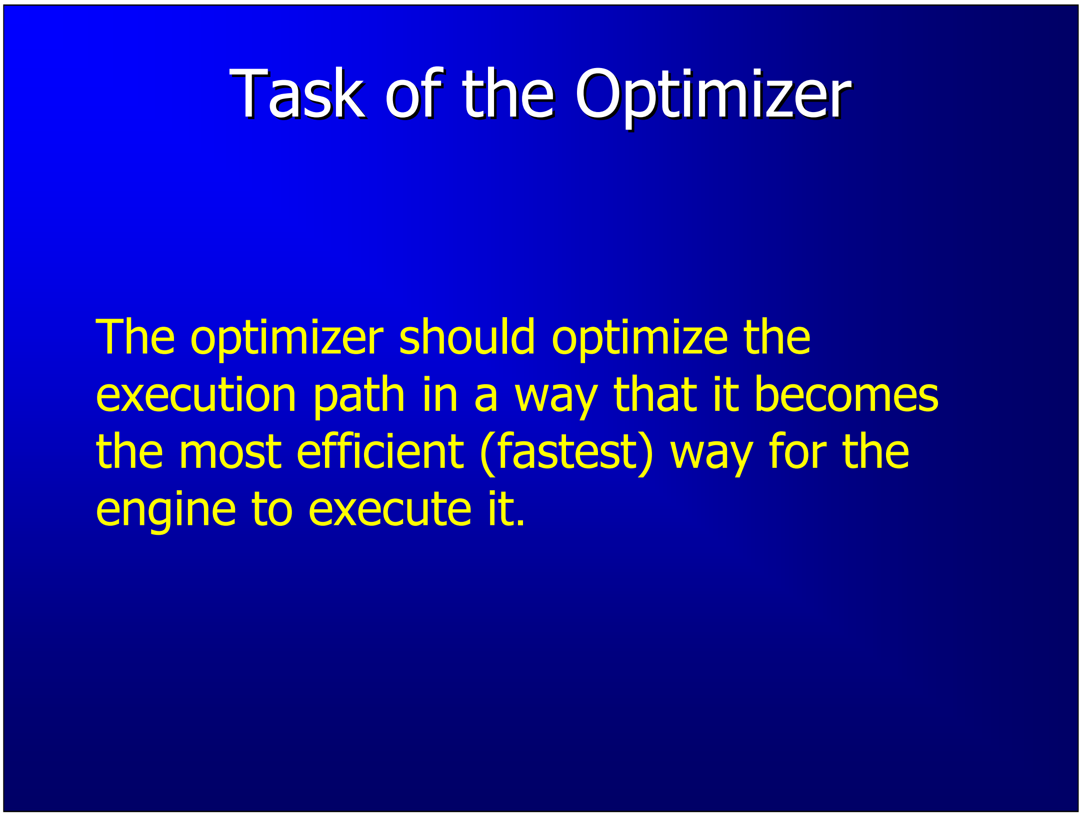
Arno Brinkman

BISIT engineering b.v.

ABVisie

firebird@abvisie.nl

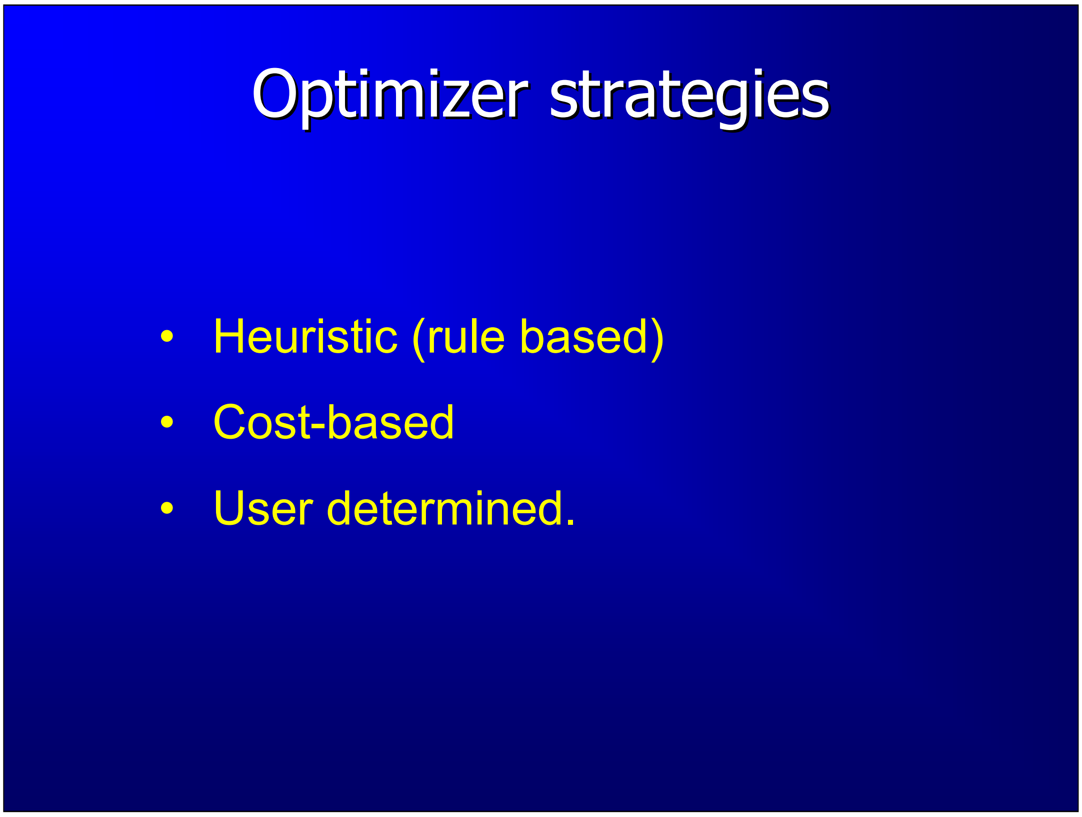
Firebird conference 2007



Task of the Optimizer

The optimizer should optimize the execution path in a way that it becomes the most efficient (fastest) way for the engine to execute it.

In fact the name Optimizer already say what it should do



Optimizer strategies

- Heuristic (rule based)
- Cost-based
- User determined.

These are the three optimizer strategies we have.

Heuristic optimizer is based on rules.

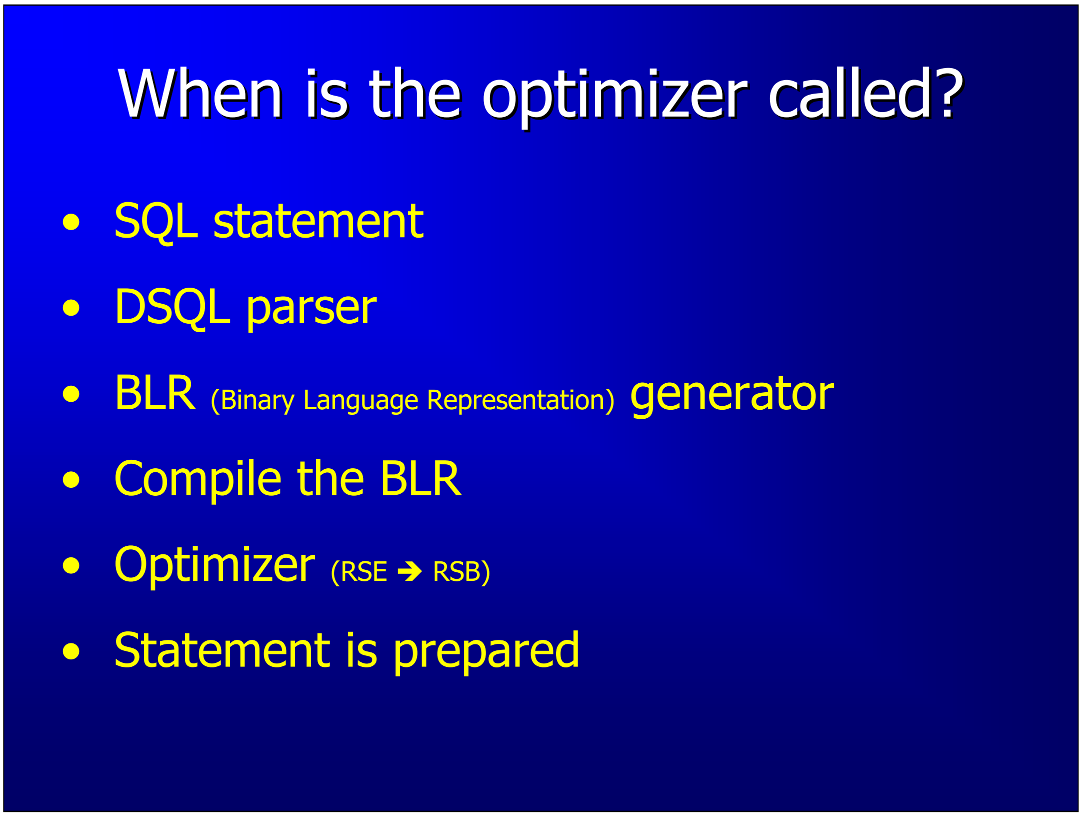
Cost-based is depending on estimated cost.

User determined optimizer in which the user controls how the execution path should look like. Such as given hints or in case of firebird a explicit PLAN which forces the execution path.

Firebird 1.5 uses cost based approach together with heuristic.

Firebird 2.0 is even some more towards the cost-based strategy.

A rule example is always preferring an PK index when there’s an equality operator on all the PK fields.



When is the optimizer called?

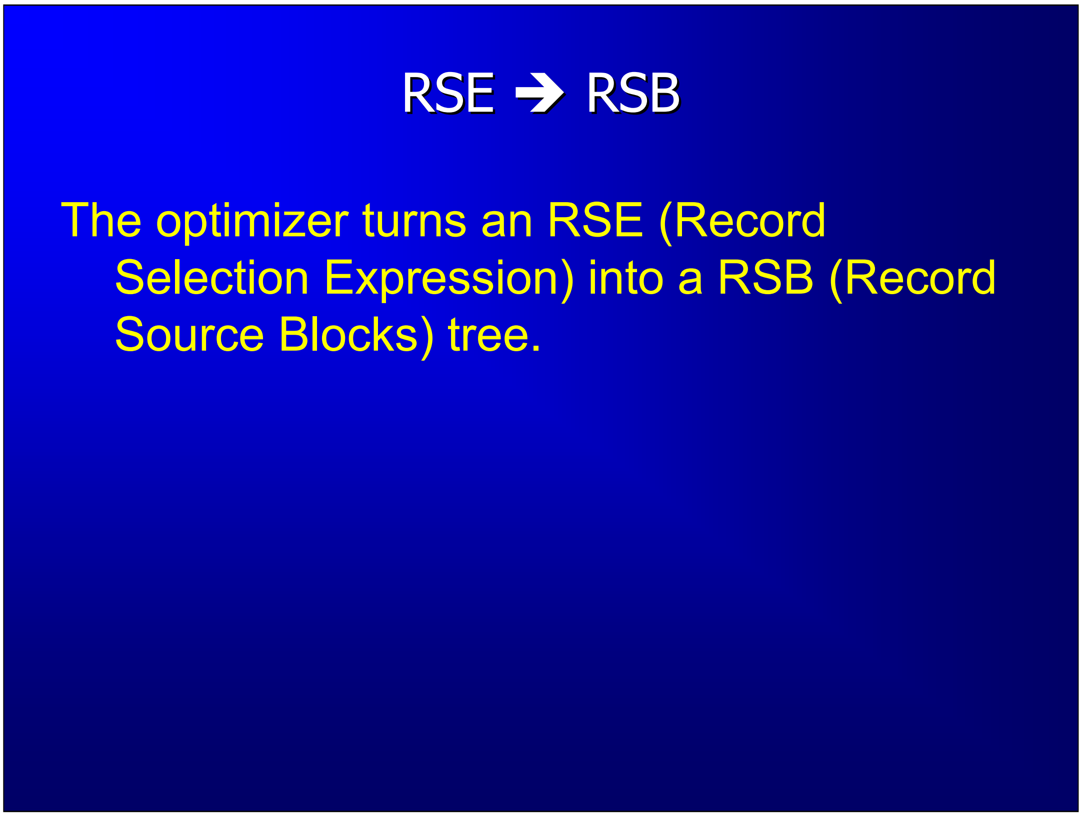
- SQL statement
- DSQL parser
- BLR (Binary Language Representation) generator
- Compile the BLR
- Optimizer (RSE ➔ RSB)
- Statement is prepared

First we have our SQL statement. This SQL statement is being parsed in the Dynamic SQL parser.
After the DSQL the BLR (Binary language representation) is generated (the op-code for the engine).
As next step the BLR is compiled into an Record Selection Expression tree were also the Optimizer is called.

The Optimizer will finally turn the RSE into a RSB tree and then the statement is prepared.

As you see many things happen before a statement is prepared.

So when you use the same statements a lot, keep them prepared in your application. Note that there is no strong binding with a transaction.

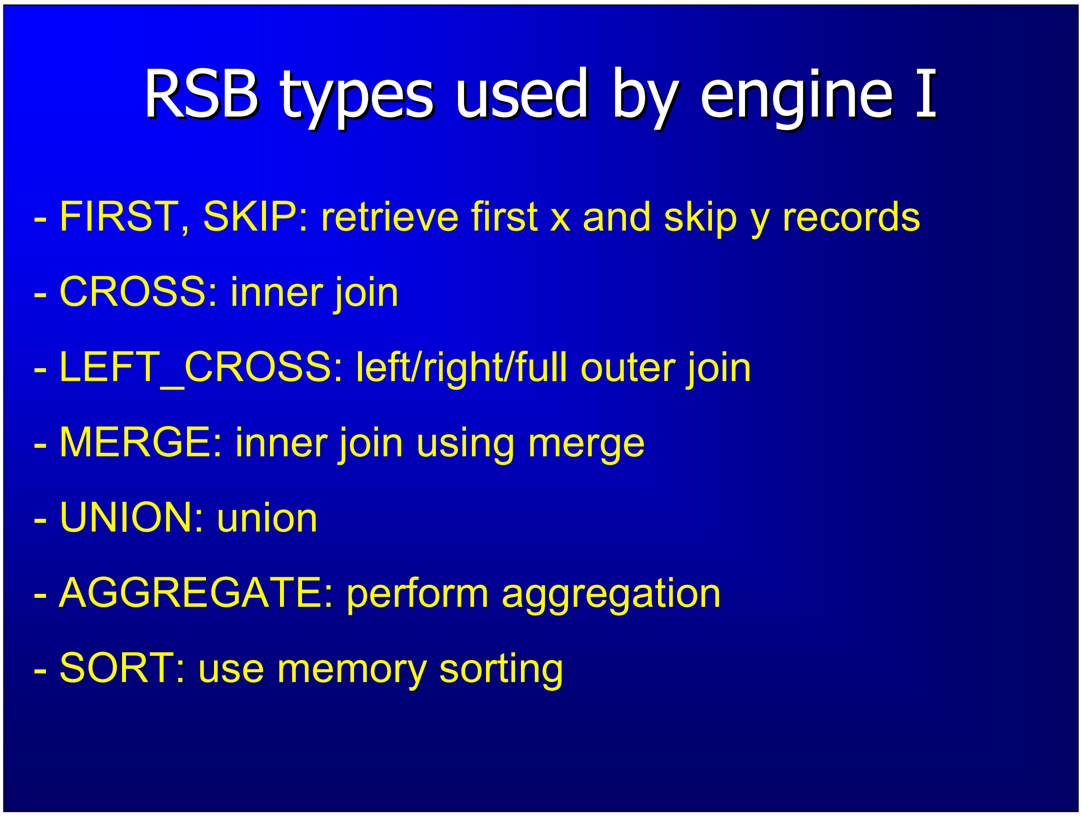


RSE → RSB

The optimizer turns an RSE (Record Selection Expression) into a RSB (Record Source Blocks) tree.

The optimizer turns a RSE (record selection expression) into a RSB (record source block) tree.

That are both tree’s with internally structures by the engine which are used to walk down a path to execute it.



RSB types used by engine I

- FIRST, SKIP: retrieve first x and skip y records
- CROSS: inner join
- LEFT_CROSS: left/right/full outer join
- MERGE: inner join using merge
- UNION: union
- AGGREGATE: perform aggregation
- SORT: use memory sorting

FIRST and SKIP: They retrieve the first X and/or skip Y records from a stream.

CROSS: That’s a nothing more then the INNER JOIN.

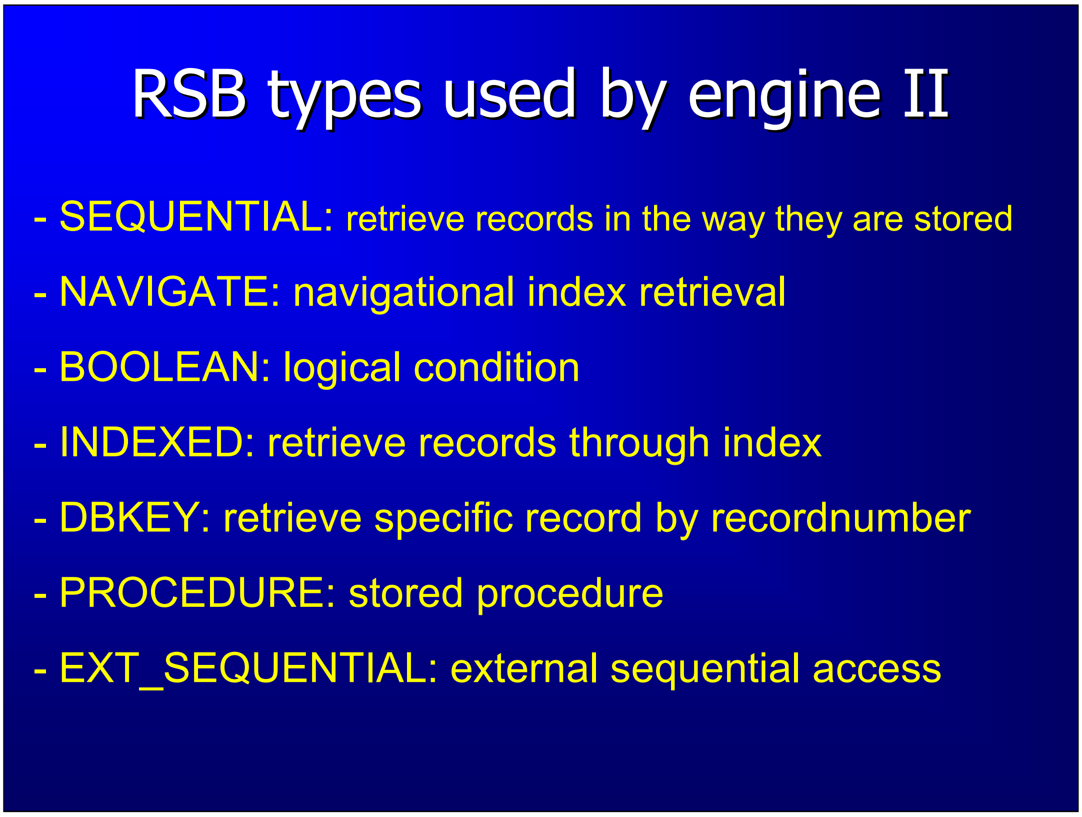
LEFT_CROSS: This can be a left/right or full outer join.

MERGE: Merge two sorted streams together (always together with SORT).

UNION: Union two or more stream.

AGGREGATE: Base stream for grouping and performing aggregate calculations (sum, total, average, …) from a stream

SORT: Sort stream. Use memory when available else swap to disk.



RSB types used by engine II

- SEQUENTIAL: retrieve records in the way they are stored
- NAVIGATE: navigational index retrieval
- BOOLEAN: logical condition
- INDEXED: retrieve records through index
- DBKEY: retrieve specific record by recordnumber
- PROCEDURE: stored procedure
- EXT_SEQUENTIAL: external sequential access

SEQUENTIAL: Retrieving the records in the way they are stored on disk.

NAVIGATE: Walking trough an index and fetching the records that belong to the index-entry.

BOOLEAN: Logical condition.

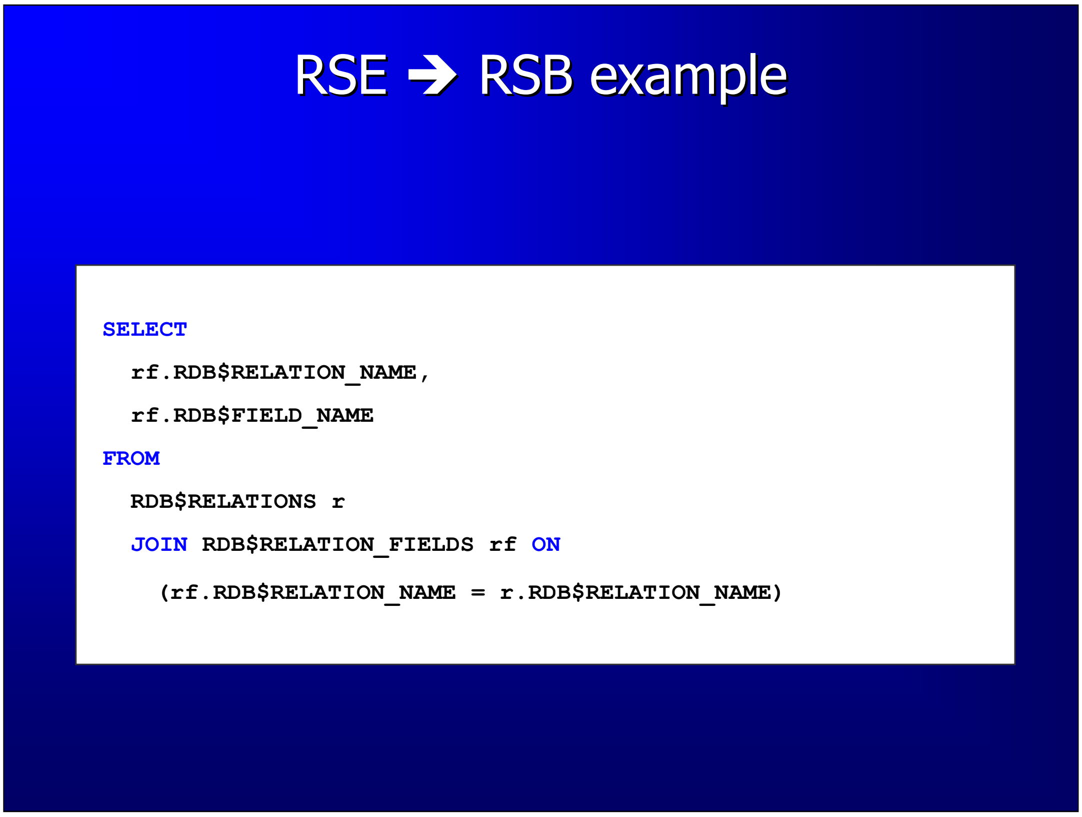
INDEXED: Filter, Retrieve records through index.

DBKEY: Retrieve explicit record by record number.

PROCEDURE: Executes stored procedure.

EXT_SEQUENTIAL: External sequential access.

There are still some other types in the engine, but they aren’t in use anymore.

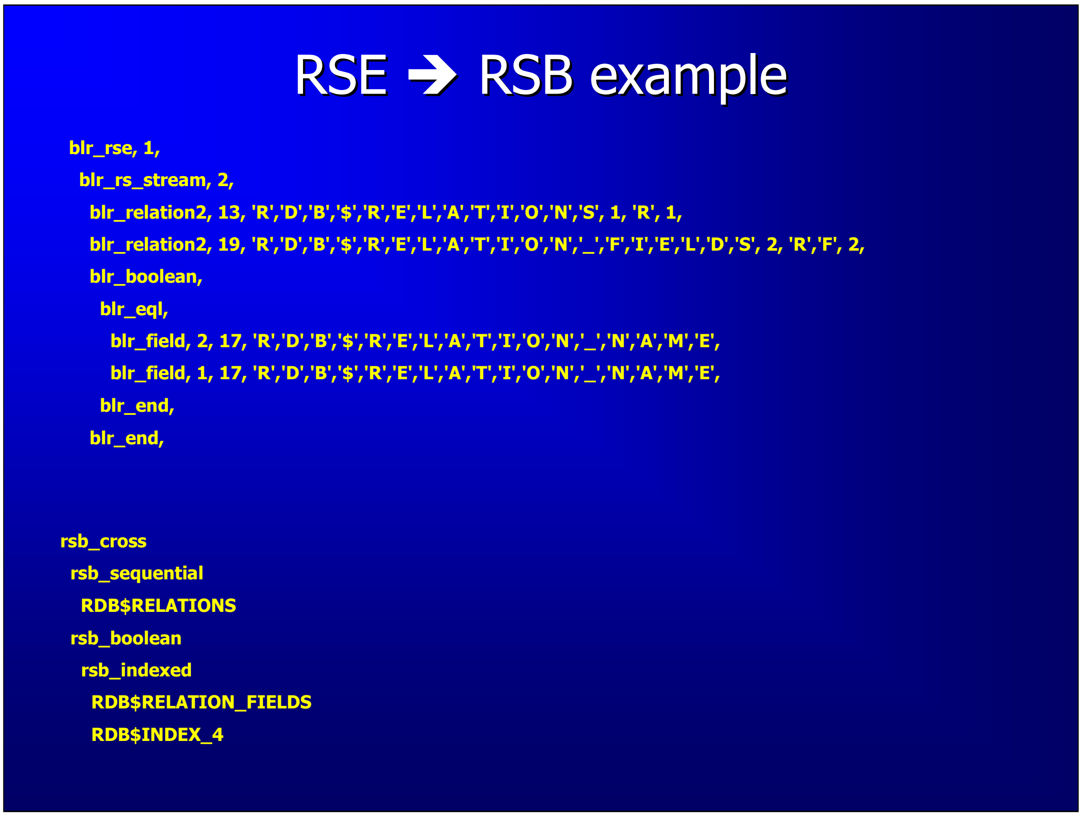


RSE → RSB example

```
SELECT
  rf.RDB$RELATION_NAME,
  rf.RDB$FIELD_NAME
FROM
  RDB$RELATIONS r
  JOIN RDB$RELATION_FIELDS rf ON
    (rf.RDB$RELATION_NAME = r.RDB$RELATION_NAME)
```

Example SQL statement i’ll use for a RSE to RSB conversion.

Assume we’re using this simple statement against the system tables.



RSE → RSB example

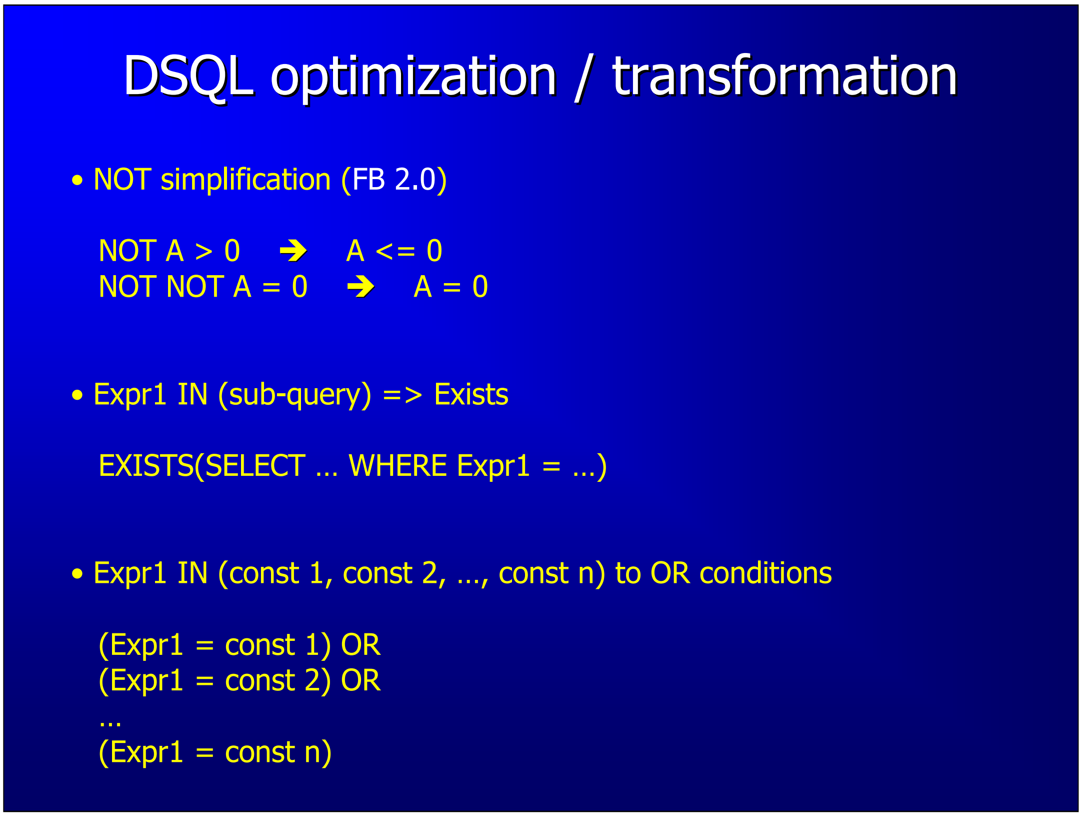
```
blr_rse, 1,
 blr_rs_stream, 2,
  blr_relation2, 13, 'R','D','B','$','R','E','L','A','T','I','O','N','S', 1, 'R', 1,
  blr_relation2, 19, 'R','D','B','$','R','E','L','A','T','I','O','N','_','F','I','E','L','D','S', 2, 'R','F', 2,
  blr_boolean,
   blr_eql,
    blr_field, 2, 17, 'R','D','B','$','R','E','L','A','T','I','O','N','_','N','A','M','E',
    blr_field, 1, 17, 'R','D','B','$','R','E','L','A','T','I','O','N','_','N','A','M','E',
   blr_end,
  blr_end,


rsb_cross
 rsb_sequential
  RDB$RELATIONS
 rsb_boolean
  rsb_indexed
   RDB$RELATION_FIELDS
   RDB$INDEX_4
```

Then the RSE part (BLR-code) of the previous statement will look as the above part in the screen. (When you create a VIEW with this statement you can see this BLR in the blob-field RDB$VIEW_BLR of RDB$RELATIONS)

The RSE goes into the optimizer and will turn it into a RSB tree as you see below.

First you see RSB_CROSS which meant JOIN (here two) streams.

The first stream has RSB_SEQUENTIAL and thus reads all records from the table RDB$RELATIONS in storage order. For every record from the first stream the RSB_BOOLEAN is executed which calls RSB_INDEXED and retrieves all records evaluated by the RDB$INDEX_4 scan on RDB$RELATION_FIELDS. For every record returned by RDB$RELATION_FIELDS the boolean condition is evaluated.



DSQL optimization / transformation

- NOT simplification (FB 2.0)

  NOT A > 0 → A <= 0
  NOT NOT A = 0 → A = 0

- Expr1 IN (sub-query) => Exists

  EXISTS(SELECT … WHERE Expr1 = …)

- Expr1 IN (const 1, const 2, …, const n) to OR conditions

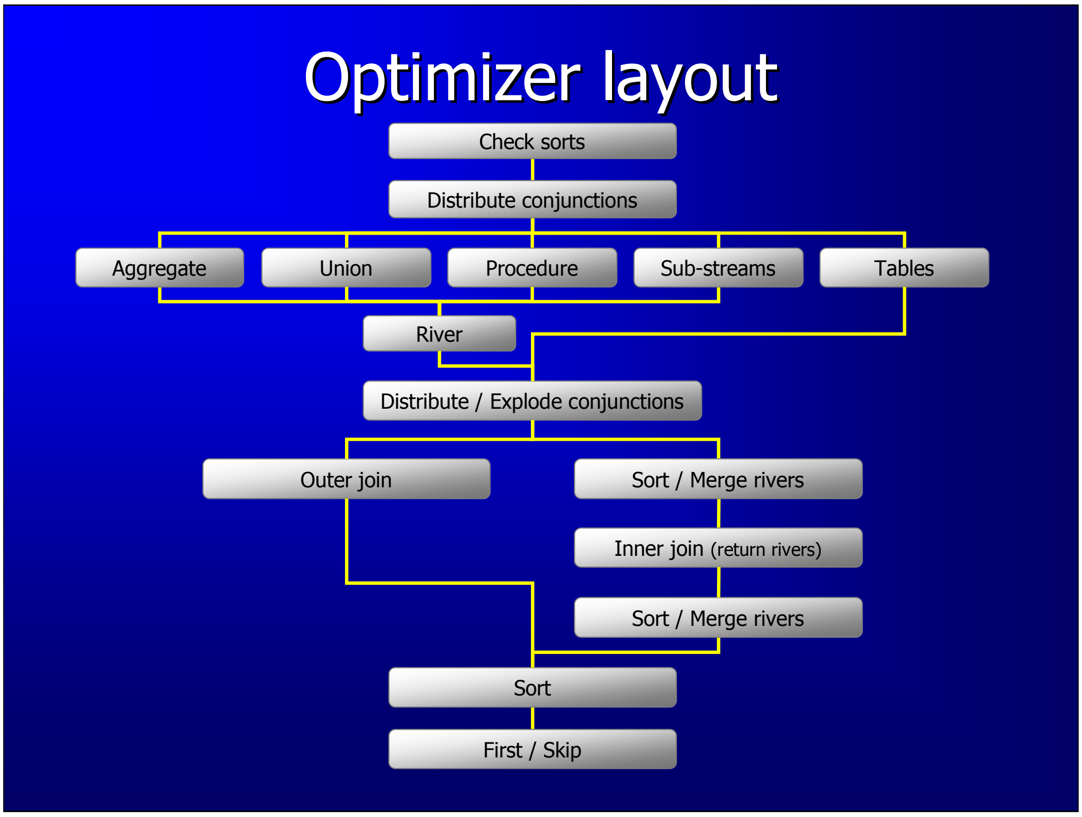
  (Expr1 = const 1) OR
  (Expr1 = const 2) OR
  …
  (Expr1 = const n)

There are a few optimizations/transformations that are done in the DSQL layer.

The NOT condition is simplified when possible, so that it can eventually use indexes (FB 2.0).

An sub-query used in the IN predicate is transformed to an EXISTS.

IN predicate with list of values is converted to a list of OR conditions



Optimizer layout

Check sorts

Distribute conjunctions

Aggregate

Union

Procedure

Sub-streams

Tables

River

Distribute / Explode conjunctions

Outer join

Sort / Merge rivers

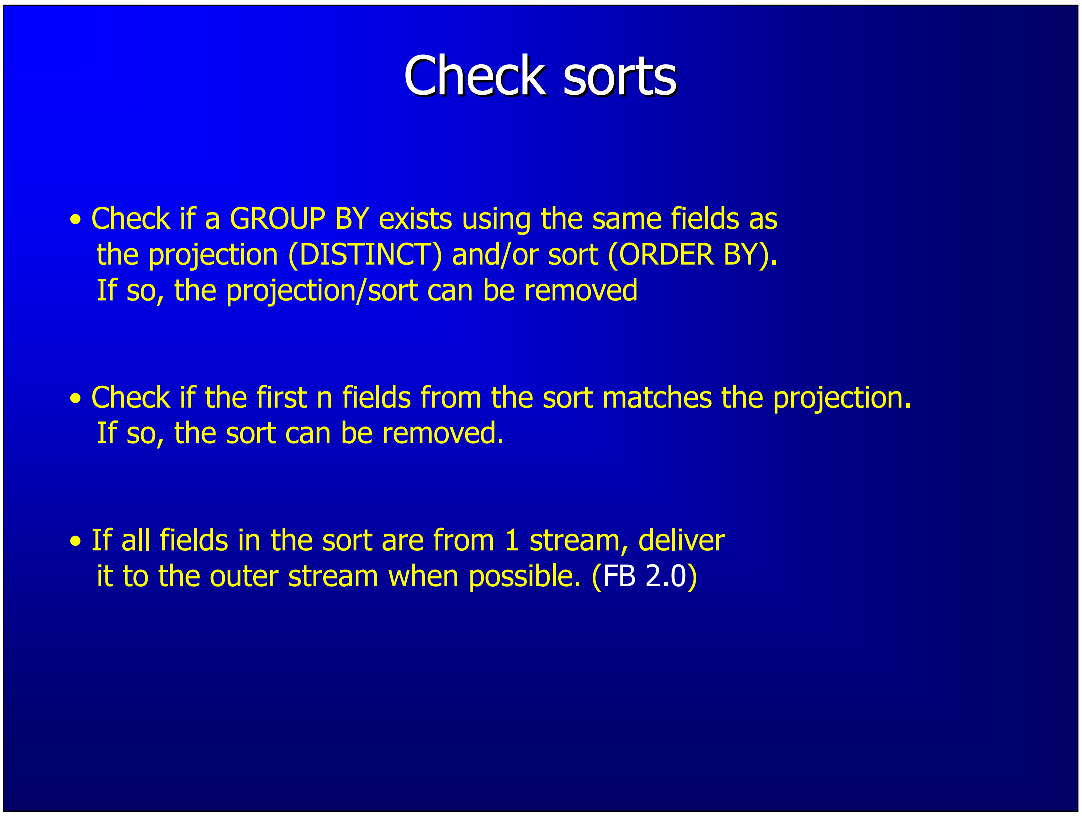
Inner join (return rivers)

Sort / Merge rivers

Sort

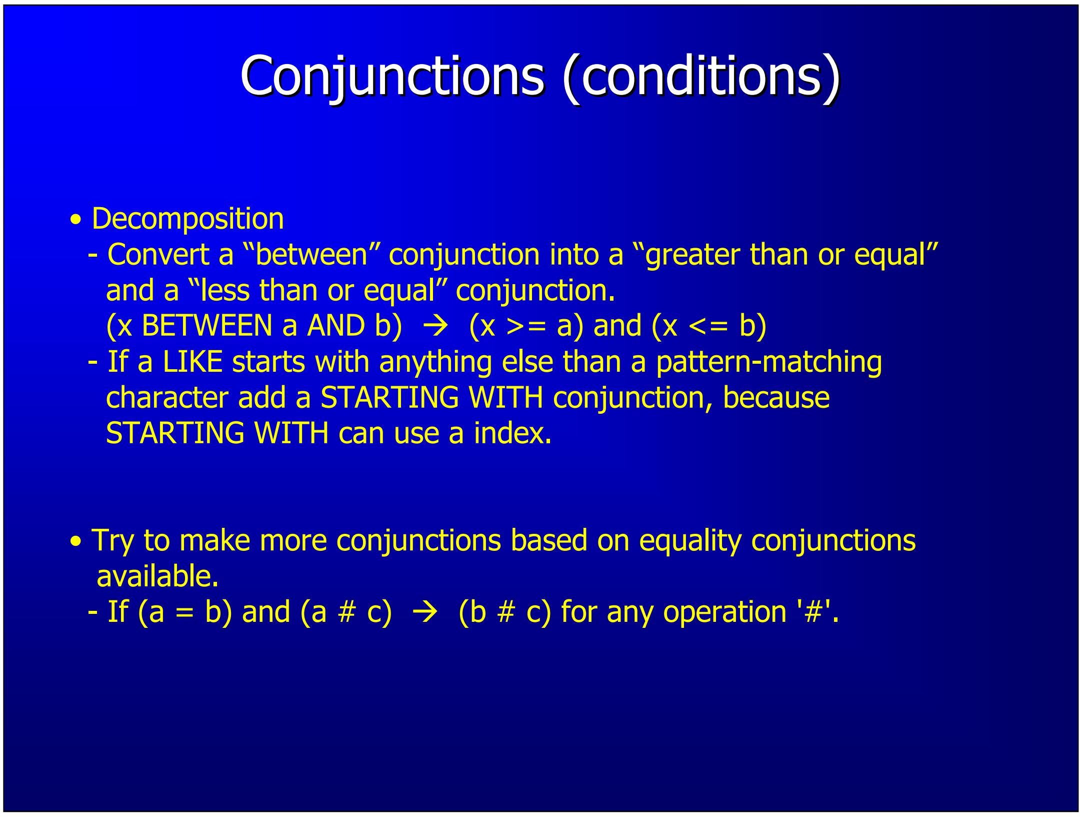
First / Skip

Simple overview of how the flow of the optimizer is internally.



Check sorts

- Check if a GROUP BY exists using the same fields as the projection (DISTINCT) and/or sort (ORDER BY). If so, the projection/sort can be removed
- Check if the first n fields from the sort matches the projection. If so, the sort can be removed.
- If all fields in the sort are from 1 stream, deliver it to the outer stream when possible. (FB 2.0)



Conjunctions (conditions)

- Decomposition
  - Convert a "between" conjunction into a "greater than or equal" and a "less than or equal" conjunction.
    (x BETWEEN a AND b) → (x >= a) and (x <= b)
  - If a LIKE starts with anything else than a pattern-matching character add a STARTING WITH conjunction, because STARTING WITH can use a index.

- Try to make more conjunctions based on equality conjunctions available.
  - If (a = b) and (a # c) → (b # c) for any operation '#'.

Decomposition:

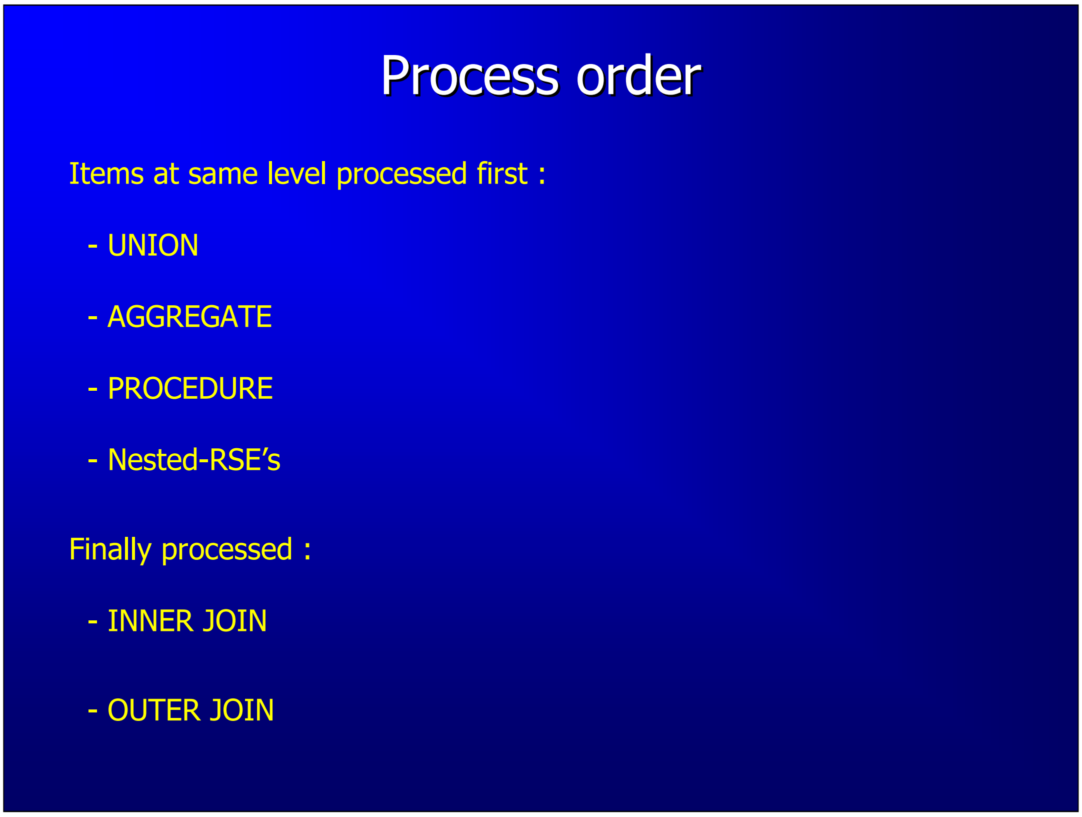
The optimizer tries to convert/expand the conjunctions so it can do more with it.

For example:

- Convert a between conjunction into a "greater than or equal" and "less than or equal" conjunction.

- We have the like: If the like starts with anything else than a pattern-matching character then a STARTING WITH conjunction is added, because only the STARTING WITH can be used with the index. The LIKE itself cannot use an index. When a LIKE is used together with a parameter (.. WHERE FieldA LIKE :parameter) never an index can be used, because at prepare time it's unknown what value will go into the parameter.

It also tries to make more conjunctions based on equality conjunctions available. This to helps binding/choosing the best index available. For example you've (a = b) and (a >= c) this will result in a new conjunction (b >= c).



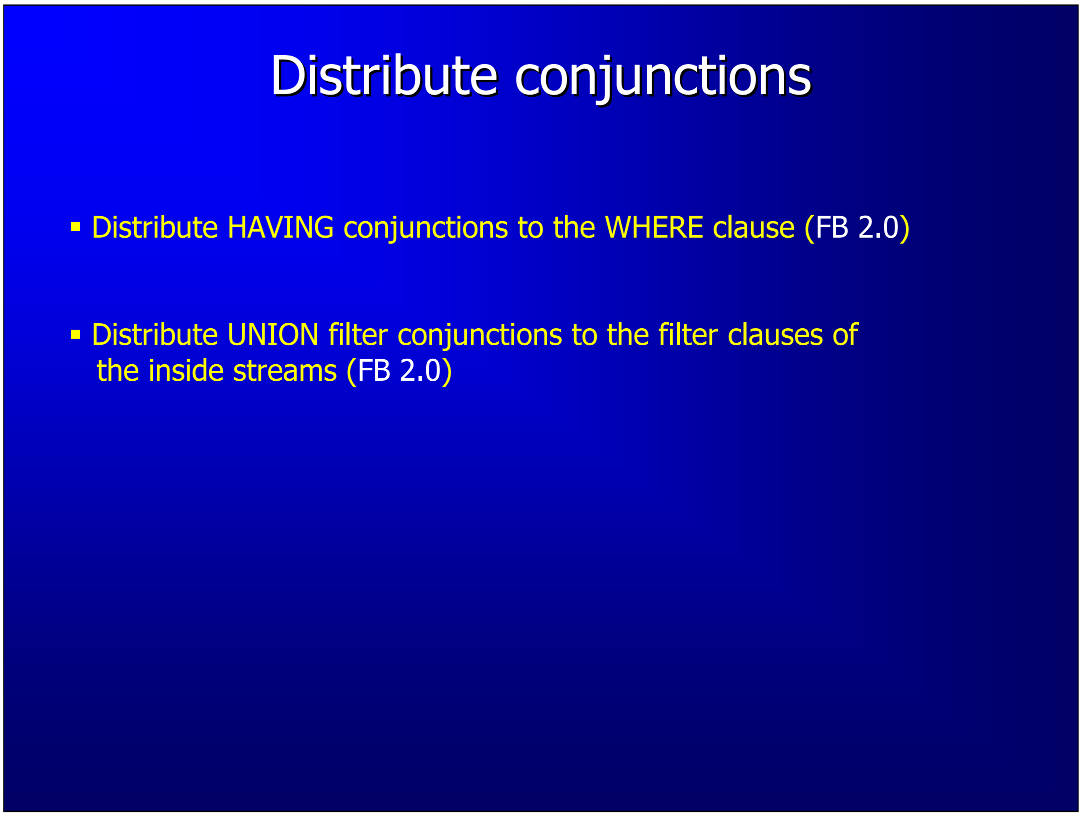
Process order

Items at same level processed first :

- UNION
- AGGREGATE
- PROCEDURE
- Nested-RSE’s

Finally processed :

- INNER JOIN
- OUTER JOIN

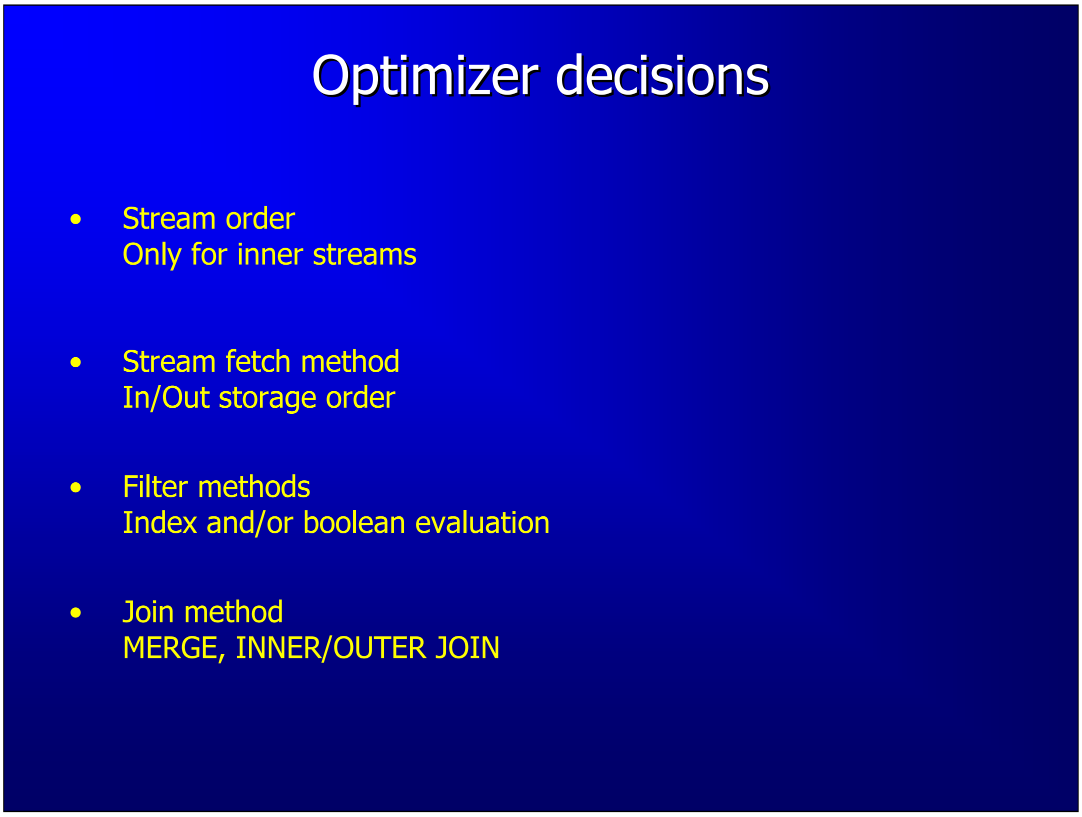


Distribute conjunctions

- Distribute HAVING conjunctions to the WHERE clause (FB 2.0)
- Distribute UNION filter conjunctions to the filter clauses of the inside streams (FB 2.0)

Distribute HAVING conjunctions to the WHERE clause (FB 2.0)
Conjunctions in the HAVING clause which contain fields that are also in the GROUP BY clause can be distributed to the WHERE clause, so that it eventually can use an index.

Distribute UNION filter conjunctions to the filter clauses of the inside streams (FB 2.0)
When an UNION is part of a VIEW and you perform a filter on the VIEW the filter is distributed to every query from the UNION.



Optimizer decisions

- Stream order
  Only for inner streams
- Stream fetch method
  In/Out storage order
- Filter methods
  Index and/or boolean evaluation
- Join method
  MERGE, INNER/OUTER JOIN

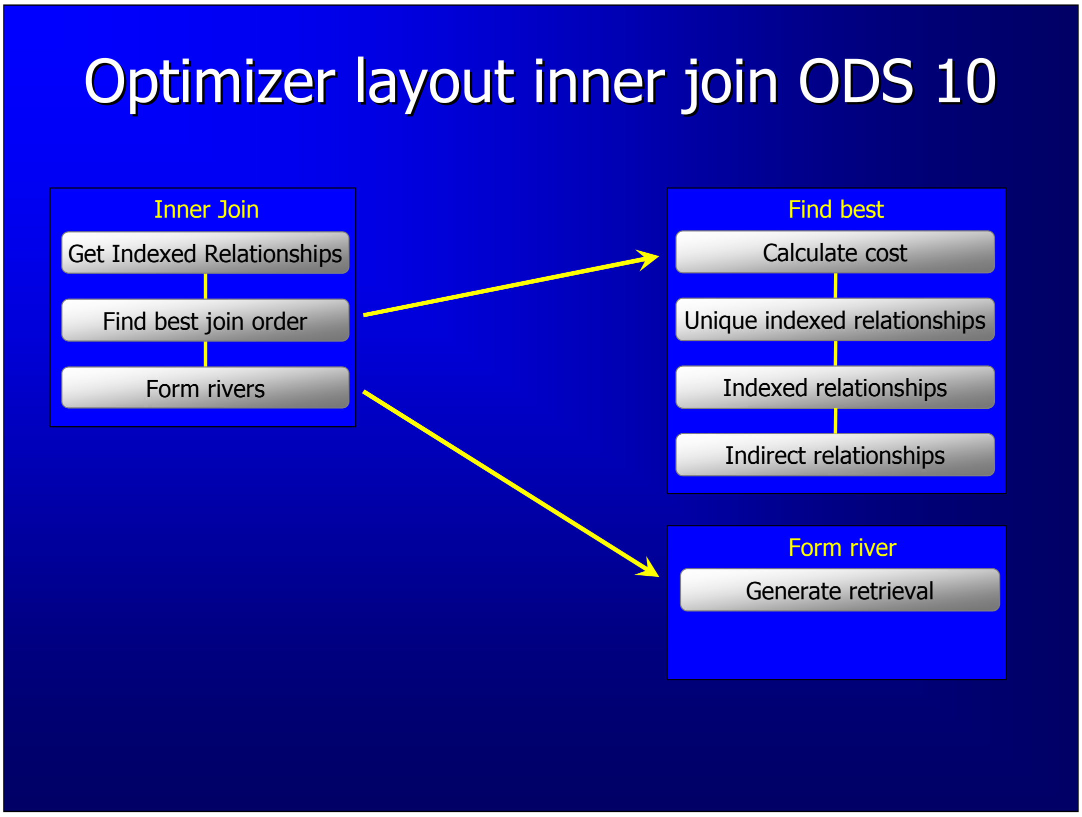
The Optimizer makes decisions about:

-The stream order for inner joins.
It decides in which order the streams should be executed after each other.

-Stream fetch method.
In or Out storage order. Which meant read it in the way it’s stored on disk (In storage order) or use an index navigation.

-Filter methods.
Index and/or Boolean evaluation.

-JOIN method.
MERGE, INNER/OUTER JOIN



Optimizer layout inner join ODS 10

Inner Join

- Get Indexed Relationships
- Find best join order
- Form rivers

Find best

- Calculate cost
- Unique indexed relationships
- Indexed relationships
- Indirect relationships

Form river

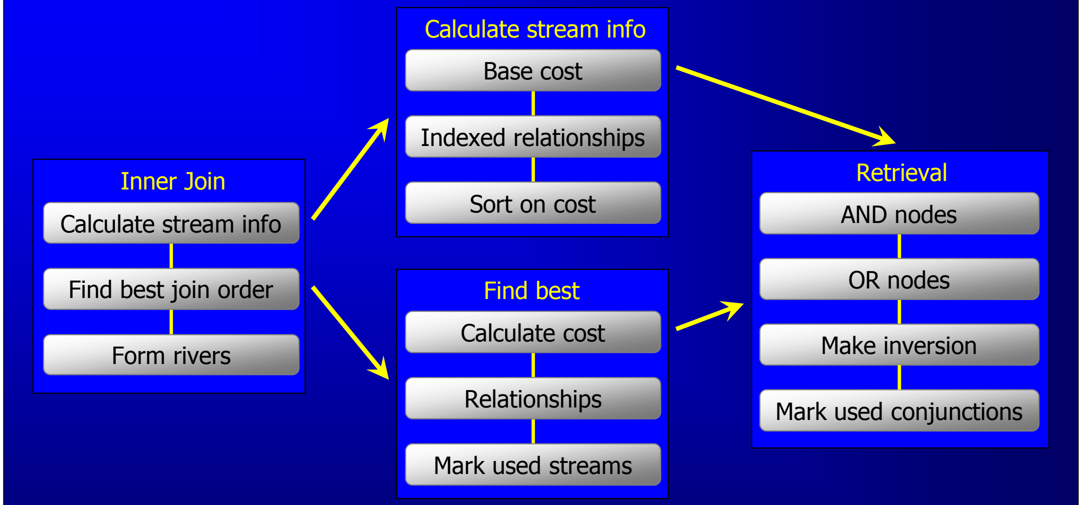
- Generate retrieval

Simplified overview of the inner join part in the optimizer for ODS 10.

First the indexed relationships between the streams are determined.
Next search for the best join order.

Finally match the conjunctions to the indexes and form rivers.

Optimizer layout inner join ODS 11



Inner Join

Calculate stream info

Find best join order

Form rivers

Calculate stream info

Base cost

Indexed relationships

Sort on cost

Find best

Calculate cost

Relationships

Mark used streams

Retrieval

AND nodes

OR nodes

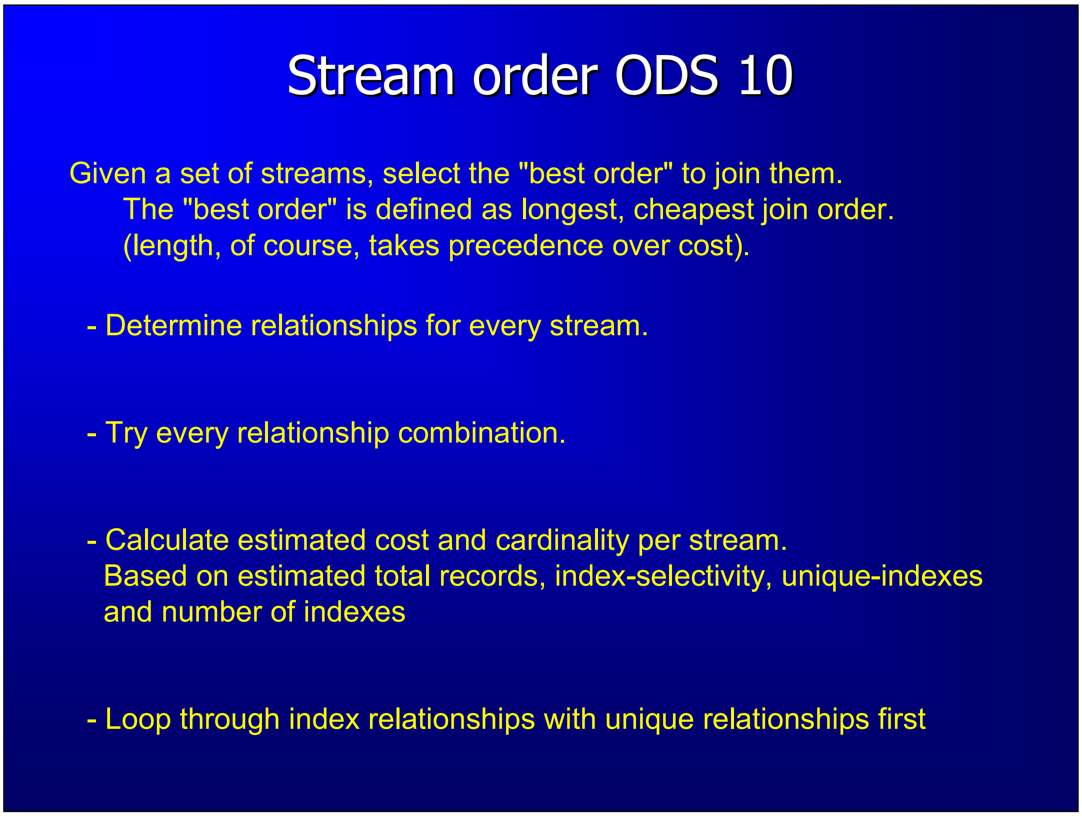
Make inversion

Mark used conjunctions

Simplified overview of the inner join part in the optimizer for ODS 11.

First the base cost / indexed relationships between the streams are calculated. Next do a walk for every stream and recursively for the next cheapest indexed relationship stream.

Finally match the conjunctions to the indexes and form rivers.



Stream order ODS 10

Given a set of streams, select the "best order" to join them.
The "best order" is defined as longest, cheapest join order.
(length, of course, takes precedence over cost).

- Determine relationships for every stream.

- Try every relationship combination.

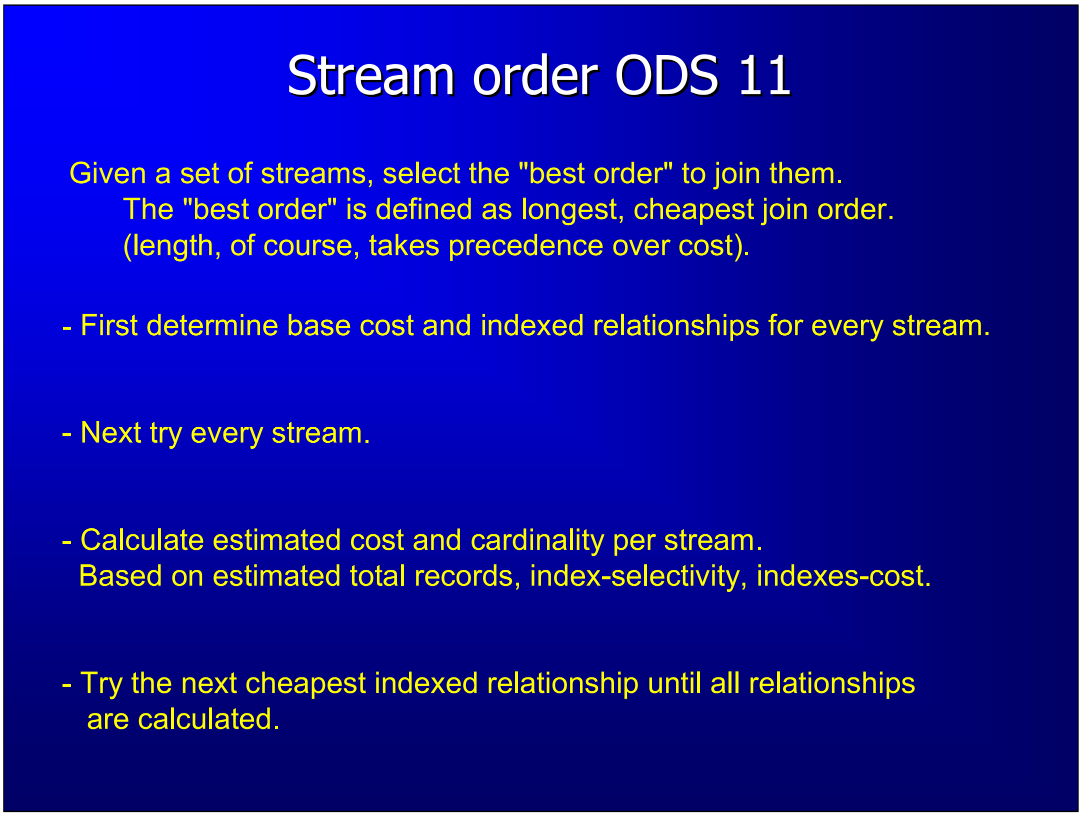
- Calculate estimated cost and cardinality per stream.
  Based on estimated total records, index-selectivity, unique-indexes and number of indexes

- Loop through index relationships with unique relationships first

First determine relationships for every stream with other streams inside the same INNER JOIN “block”.
Next try every relationship combination.
Calculate estimated cost and cardinality per stream.
This is based on estimated total records, index-selectivity, unique-indexes and number of indexes

Loop through direct index relationships with direct unique relationships first.
This is done because the unique relationships are the most interesting/cheapest relationships.

Finally check for indirect indexed relationships.



Stream order ODS 11

Given a set of streams, select the "best order" to join them.
The "best order" is defined as longest, cheapest join order.
(length, of course, takes precedence over cost).

- First determine base cost and indexed relationships for every stream.

- Next try every stream.

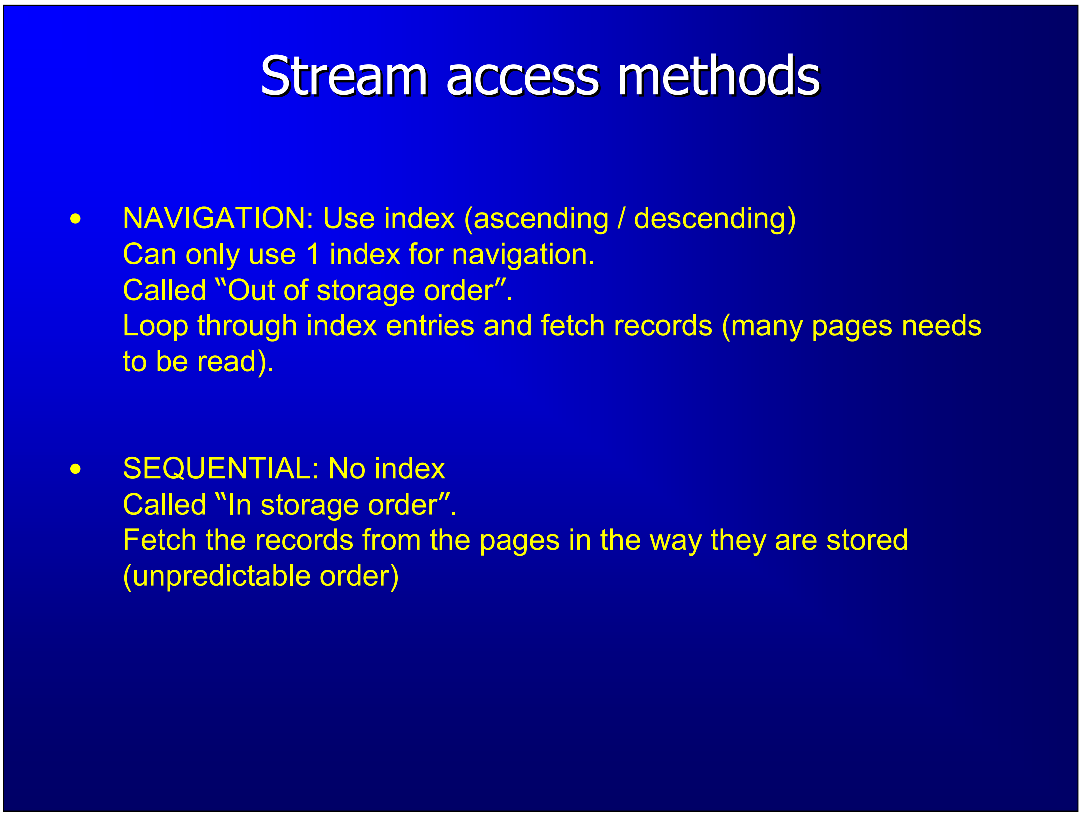
- Calculate estimated cost and cardinality per stream.
  Based on estimated total records, index-selectivity, indexes-cost.

- Try the next cheapest indexed relationship until all relationships are calculated.

First determine base cost and indexed relationships for every stream.

Next try every stream.
Calculate estimated cost and cardinality per stream at each position.
This is based on estimated total records, index-selectivity, index-cost.

Try the next cheapest indexed relationship until all relationships are calculated.



Stream access methods

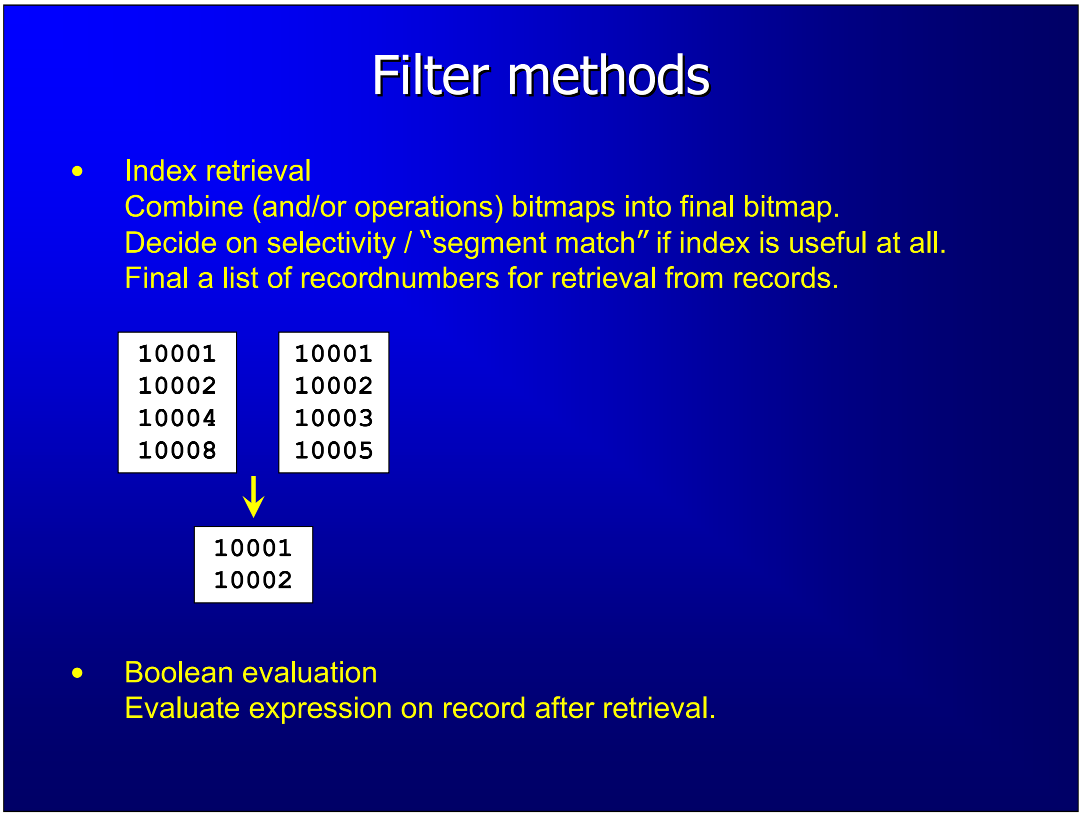
- NAVIGATION: Use index (ascending / descending)
  Can only use 1 index for navigation.
  Called "Out of storage order".
  Loop through index entries and fetch records (many pages needs to be read).

- SEQUENTIAL: No index
  Called "In storage order".
  Fetch the records from the pages in the way they are stored (unpredictable order)

We've two types of stream access methods:

-NAVIGATION
Using an ascending/descending index.
Can only use 1 index for navigation and is called "Out of storage order".
Loop through the index entries and fetch the records.
This causes many data-page reads, because in most cases for every record a new data-page must be read.
Fastest way when you want to fetch only a few records by order (FIRST X) from a huge table.

-SEQUENTIAL
No index and called "In storage order".
Fetch the records in the way the are stored on disk. This is thus in an unpredictable order.
Fastest way when you want to fetch a large piece of the table.



Filter methods

- Index retrieval
  Combine (and/or operations) bitmaps into final bitmap.
  Decide on selectivity / "segment match" if index is useful at all.
  Final a list of recordnumbers for retrieval from records.

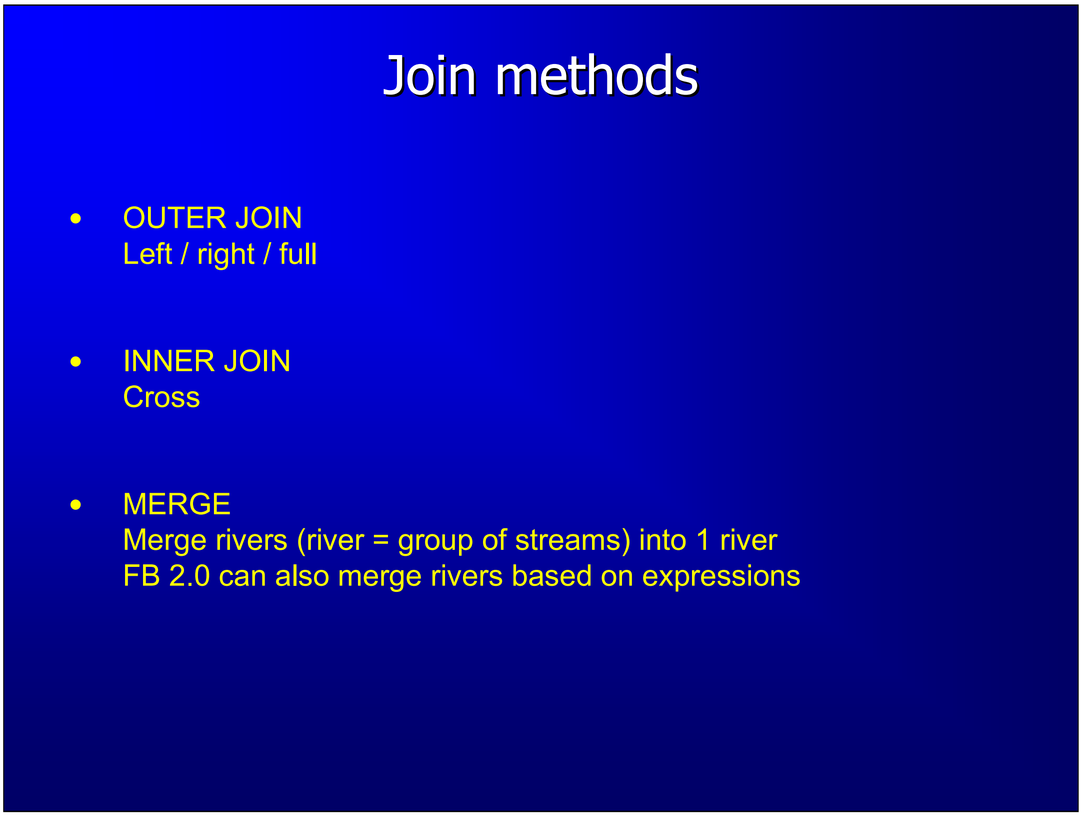
10001
10002
10004
10008

10001
10002
10003
10005

↓

10001
10002

- Boolean evaluation
  Evaluate expression on record after retrieval.



Join methods

- OUTER JOIN
  Left / right / full
- INNER JOIN
  Cross
- MERGE
  Merge rivers (river = group of streams) into 1 river
  FB 2.0 can also merge rivers based on expressions

OUTER JOIN: Left / right / full outer join.

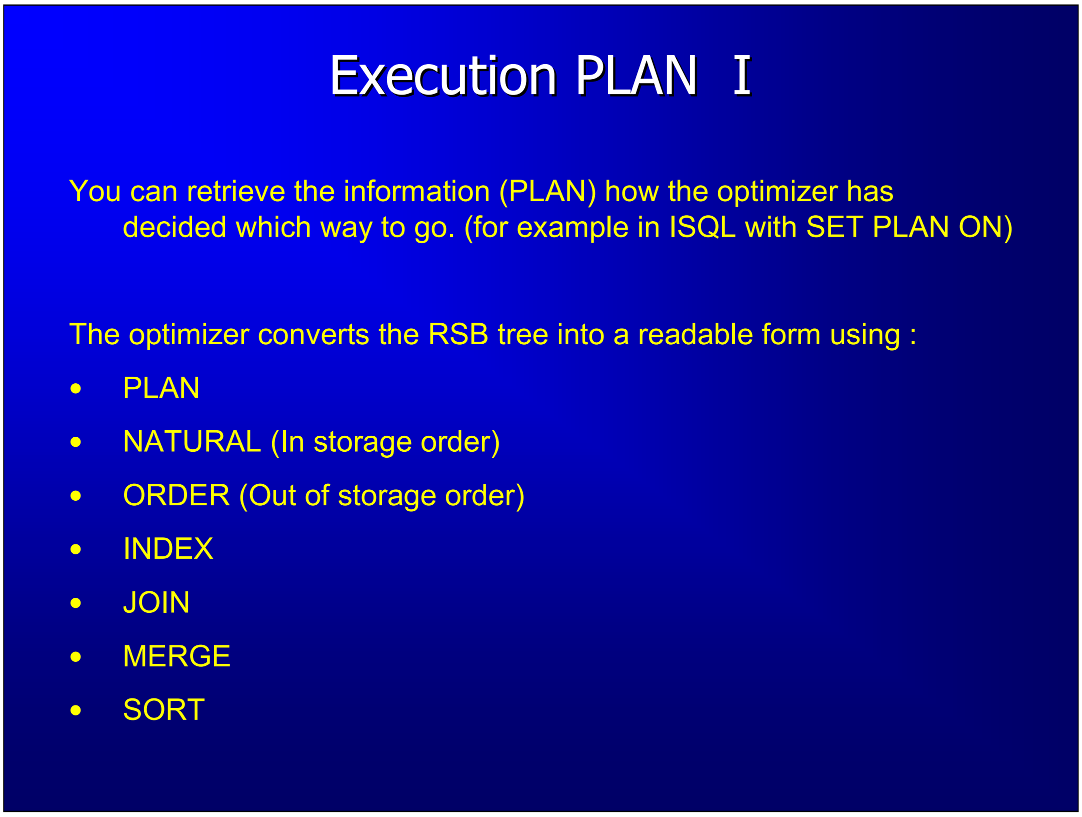
INNER JOIN: Cross, join 2 or more streams.

MERGE: Merge rivers (a river represent multiple streams) into 1 river and does only work for INNER JOINs.

When there’s a relationship between two streams, but no index could be used. It will try to sort both streams and merge them together.

FB2 can also merge streams based on expressions, such as:

SELECT * FROM Table1 t1 JOIN Table2 t2 ON (t2.ID + 100 = t1.ID)



Execution PLAN I

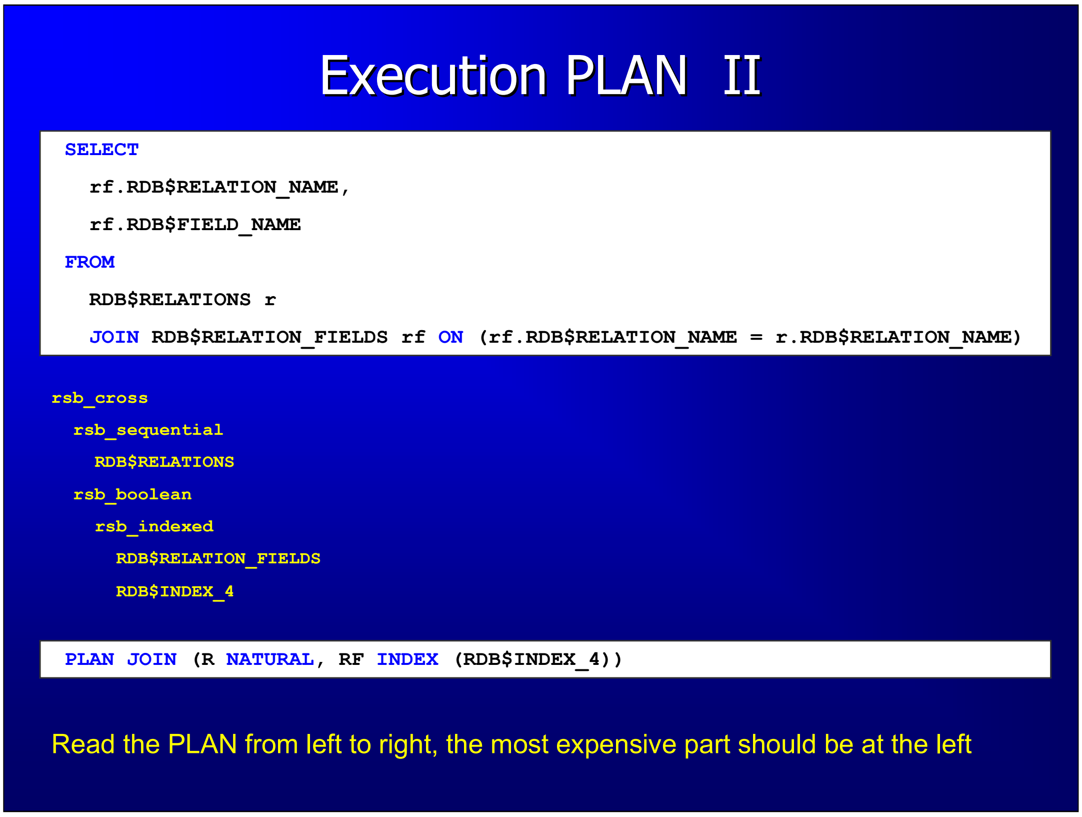
You can retrieve the information (PLAN) how the optimizer has decided which way to go. (for example in ISQL with SET PLAN ON)

The optimizer converts the RSB tree into a readable form using :

- PLAN
- NATURAL (In storage order)
- ORDER (Out of storage order)
- INDEX
- JOIN
- MERGE
- SORT

Almost every database tool can give you the PLAN from a query. Some give you even a nice graphical screen with some hints about the PLAN.

When asking for a PLAN the optimizer converts the RSB tree into a readable format using the keywords:
- PLAN (Start of a RSB tree)
- NATURAL (Sequential, In storage order)
- ORDER (Navigation, Out of storage order)
- INDEX (Retrieval by bitmap)
- JOIN (Inner/left/right/full join streams)
- MERGE (Merge two rivers)
- SORT (Sorting stream)



Execution PLAN II

```
SELECT
  rf.RDB$RELATION_NAME,
  rf.RDB$FIELD_NAME
FROM
  RDB$RELATIONS r
  JOIN RDB$RELATION_FIELDS rf ON (rf.RDB$RELATION_NAME = r.RDB$RELATION_NAME)
```

```
rsb_cross
  rsb_sequential
    RDB$RELATIONS
  rsb_boolean
    rsb_indexed
      RDB$RELATION_FIELDS
      RDB$INDEX_4
```

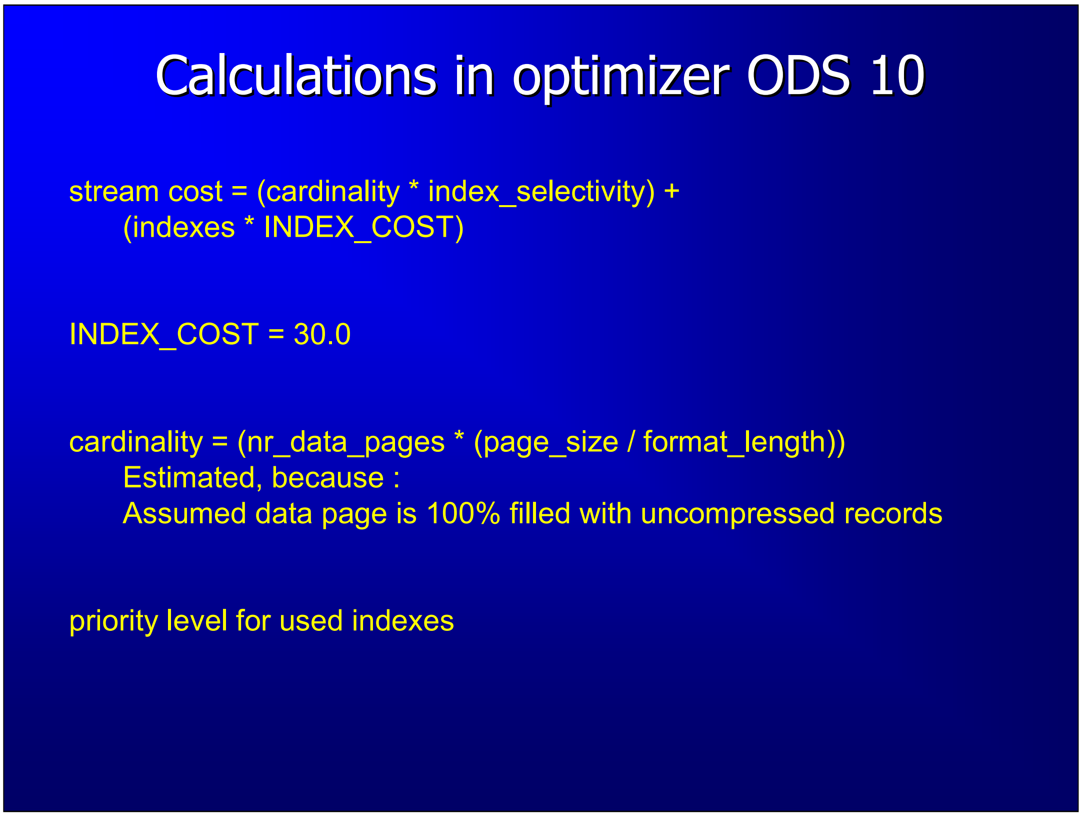
```
PLAN JOIN (R NATURAL, RF INDEX (RDB$INDEX_4))
```

Read the PLAN from left to right, the most expensive part should be at the left

Let’s took the same example as used in a slide before.

At the top the statement with below the RSB tree used internally.
Under the RSB tree you see the readable PLAN output made by the optimizer.
Read the PLAN from left to right and you’ll notice that it has the same “nodes” as the RSB tree from top to bottom.

Things at the left are thus processed first and at the right as last. As you understand a NATURAL having on the right side is mostly a expensive thing, because that will read the whole stream for every time called.



Calculations in optimizer ODS 10

stream cost = (cardinality * index_selectivity) +
(indexes * INDEX_COST)

INDEX_COST = 30.0

cardinality = (nr_data_pages * (page_size / format_length))
Estimated, because :
Assumed data page is 100% filled with uncompressed records

priority level for used indexes

Stream cost = (cardinality * index_selectivity) + (indexes * INDEX_COST)

INDEX_COST is an hard coded value of 30.0

Indexes is the number of indexes used by the stream

Cardinality = (nr_data_pages * (page_size / format_length_record))

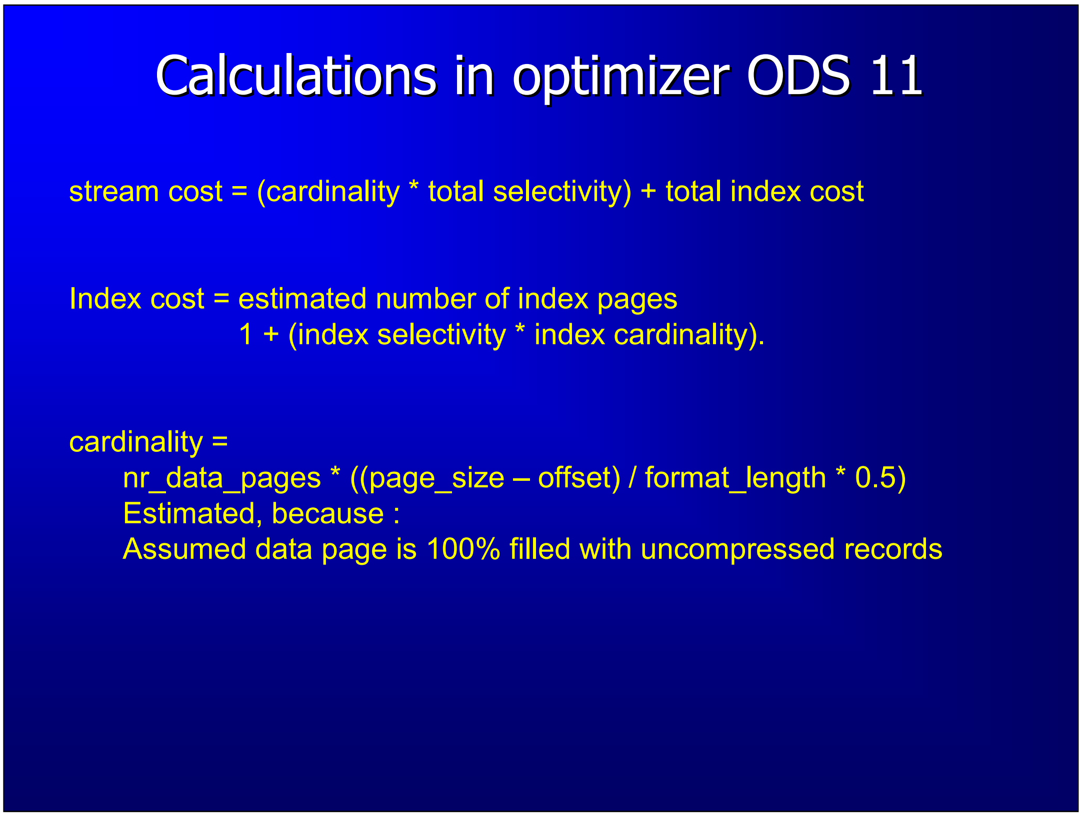
Nr_data_pages is the number of data pages for the table.
format_length_record = Uncompressed size for the record.

This is estimated, because it’s assumed that all data pages are filled for 100 percent and the records aren’t compressed.

In FB1.5 a priority level for indexes is introduced.

The priority level is based on :

- How many equality conjunctions match against the segments.
- How many conjunctions can be matched against an index.
- How many segments are matched.



Calculations in optimizer ODS 11

stream cost = (cardinality * total selectivity) + total index cost

Index cost = estimated number of index pages
1 + (index selectivity * index cardinality).

cardinality =
nr_data_pages * ((page_size – offset) / format_length * 0.5)
Estimated, because :
Assumed data page is 100% filled with uncompressed records

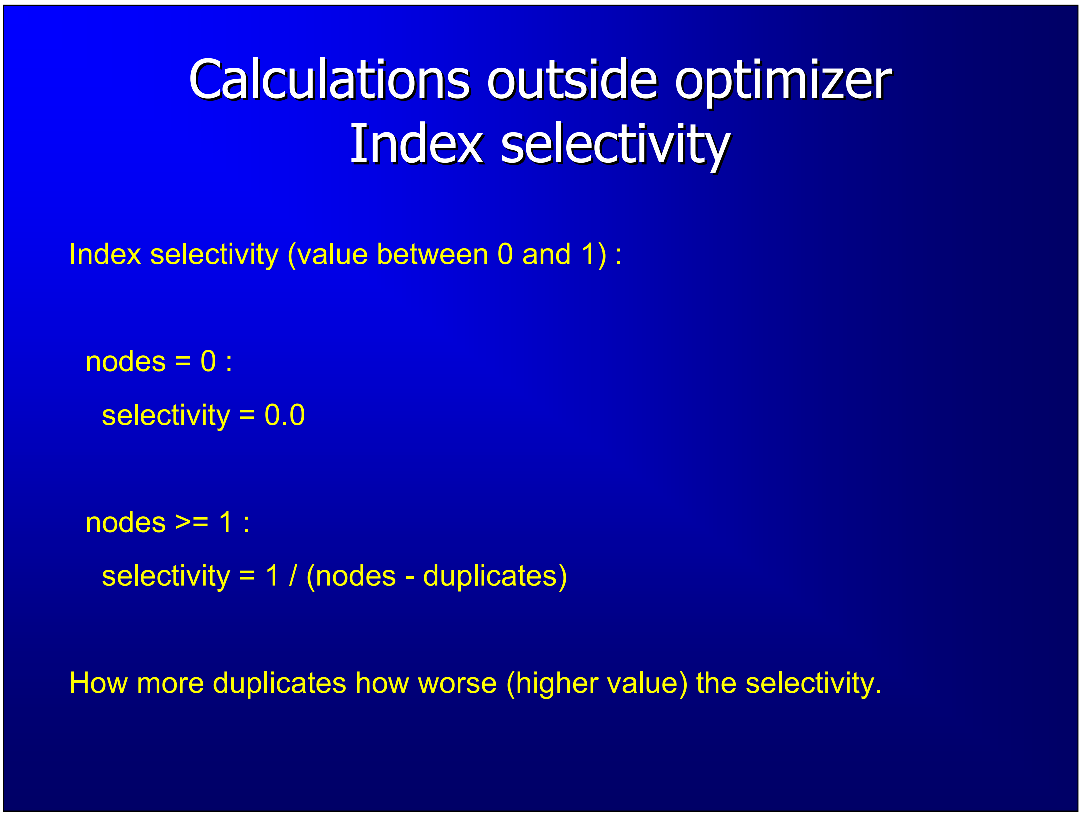
Stream cost = (cardinality * index_selectivity) + index cost

Cardinality = nr_data_pages * ((page_size – offset) / (min_rec_size + format_length_record * 0.5))

nr_data_pages is the number of data pages for the table.
format_length_record = Uncompressed size for the record.

offset = Header space used in page.
min_rec_size = Minimum size a record always will use.
Index cost = estimated number of index pages 1 + (selectivity * index cardinality).

This is estimated, because it’s assumed that all data pages are filled for 100 percent and the records aren’t compressed.



Calculations outside optimizer
Index selectivity

Index selectivity (value between 0 and 1) :

nodes = 0 :

selectivity = 0.0

nodes >= 1 :

selectivity = 1 / (nodes - duplicates)

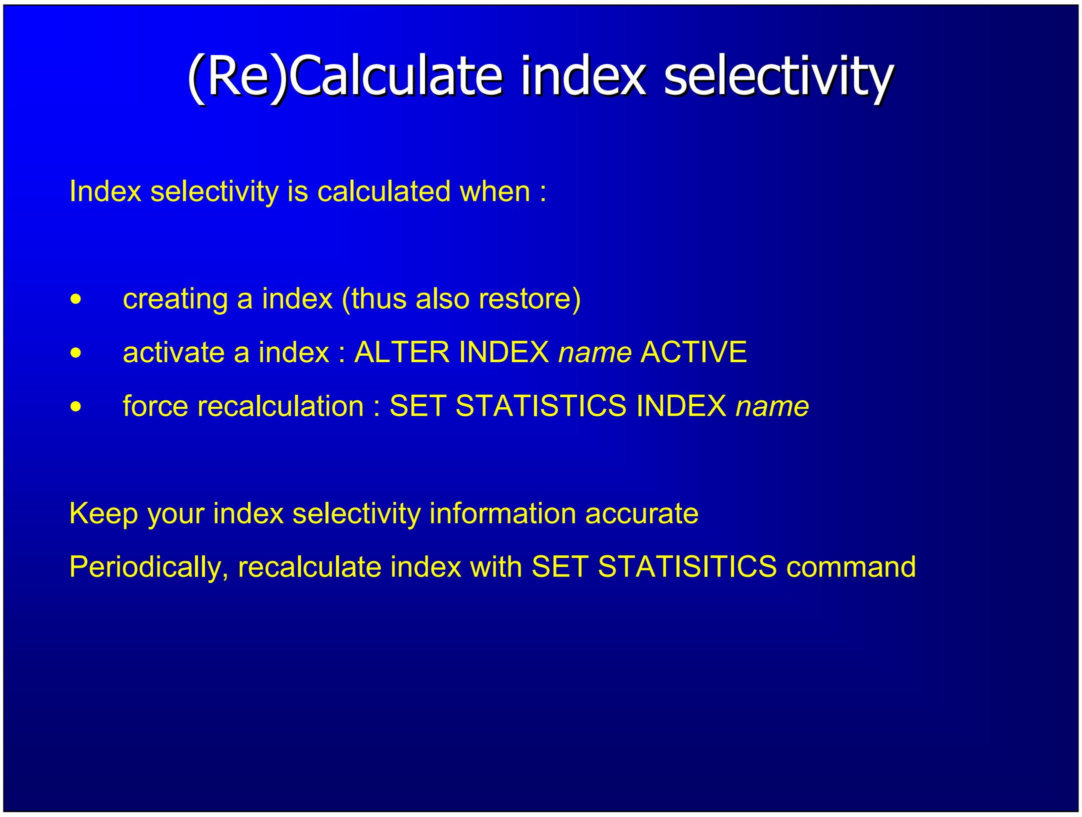
How more duplicates how worse (higher value) the selectivity.

The index selectivity is calculated outside the optimizer and belongs to the index. Selectivity is a value between 0 and 1.
If there are no nodes the selectivity is zero else the selectivity is calculated by:

Selectivity = 1 / (nodes – duplicates)

How higher the value is near to 1 the more duplicate nodes are in the index (selectivity's compared from indexes on same table). Note that the number of nodes could be higher as the number of records in the table, because the index can hold nodes which point to records which aren't visible for the current transaction. Need to be garbage collected for example.

In ODS 11 the selectivity is also calculated per segment which makes calculations with compound indexes more accurate.



(Re)Calculate index selectivity

Index selectivity is calculated when :

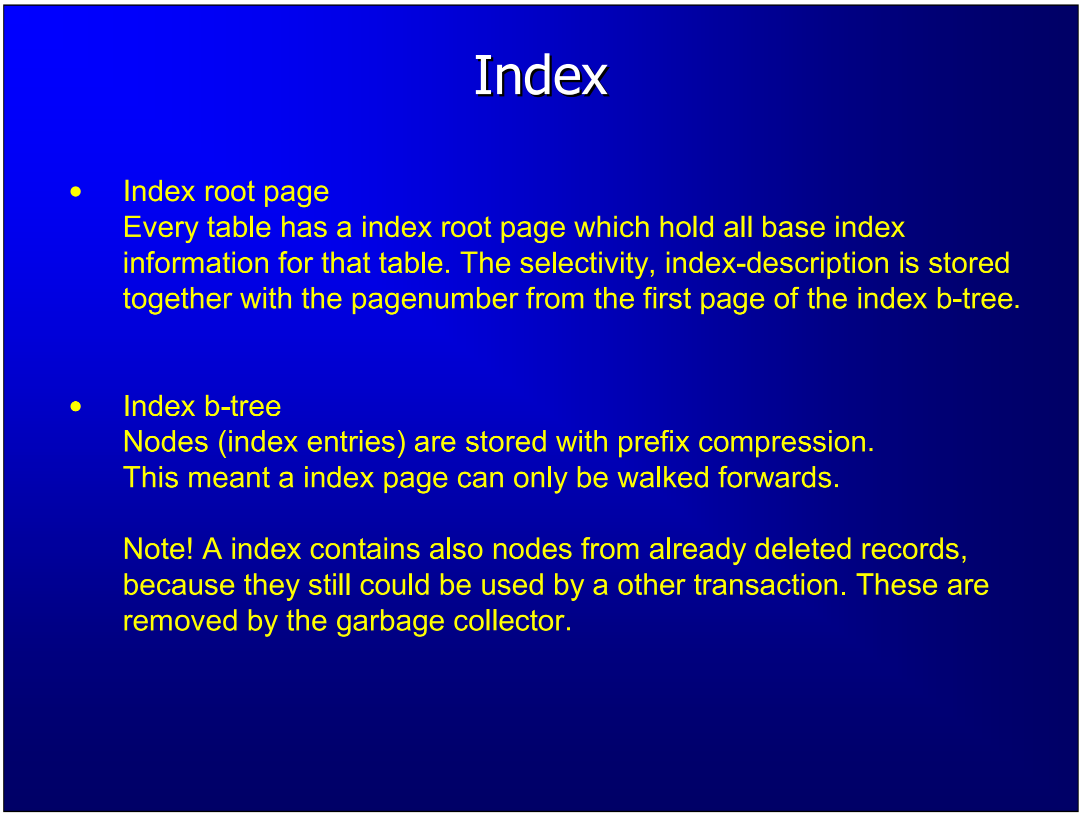
- creating a index (thus also restore)
- activate a index : ALTER INDEX *name* ACTIVE
- force recalculation : SET STATISTICS INDEX *name*

Keep your index selectivity information accurate

Periodically, recalculate index with SET STATISITICS command

Periodically, recalculate index with the SET STATISTICS command.

Also recalculate always all indexes that belong to a table, so that the selectivity's between the indexes are accurate when compared to each other.

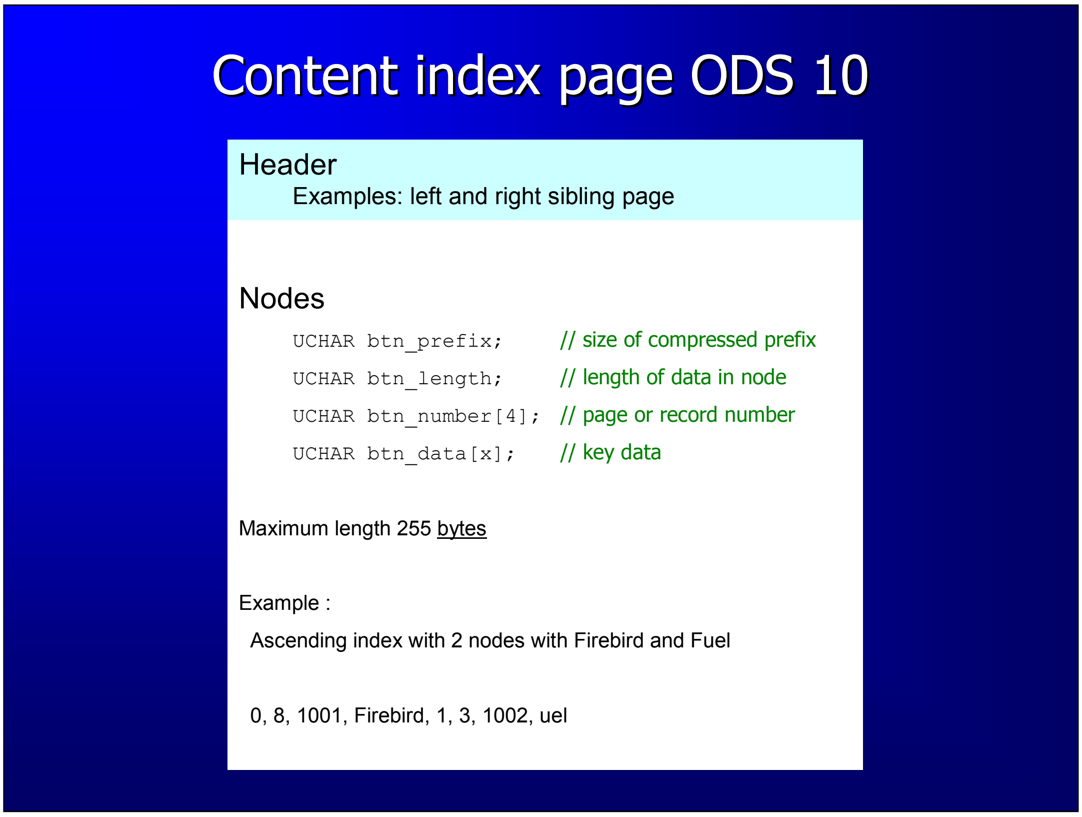


Index

- Index root page
  Every table has a index root page which hold all base index information for that table. The selectivity, index-description is stored together with the pagenumber from the first page of the index b-tree.

- Index b-tree
  Nodes (index entries) are stored with prefix compression.
  This meant a index page can only be walked forwards.

  Note! A index contains also nodes from already deleted records, because they still could be used by a other transaction. These are removed by the garbage collector.



Content index page ODS 10

Header

Examples: left and right sibling page

Nodes

```
UCHAR btn_prefix;       // size of compressed prefix
UCHAR btn_length;       // length of data in node
UCHAR btn_number[4];    // page or record number
UCHAR btn_data[x];      // key data
```

Maximum length 255 bytes

Example :

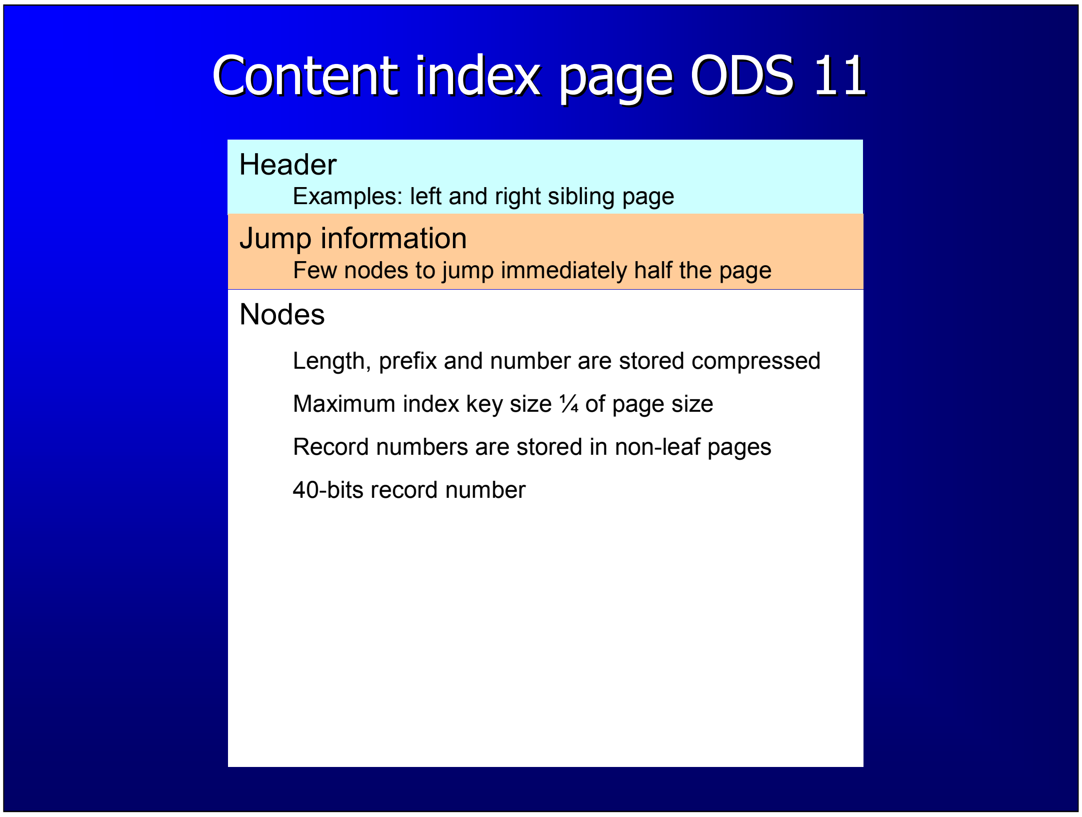
Ascending index with 2 nodes with Firebird and Fuel

0, 8, 1001, Firebird, 1, 3, 1002, uel

Maximum index size for Firebird 1.5 and lower is 255 bytes, but note that collations can use 3 bytes for only 1 character and thus decreases the size to a maximum of 84 characters. Beside that when a compound index is used also for every 4 bytes an 1 byte “segment-marker” is added.

A nice key size calculator can be found on the site from Ivan Prenosil:

http://www.volny.cz/iprenosil/interbase/ip_ib_indexcalculator.htm



Content index page ODS 11

Header

Examples: left and right sibling page

Jump information

Few nodes to jump immediately half the page

Nodes

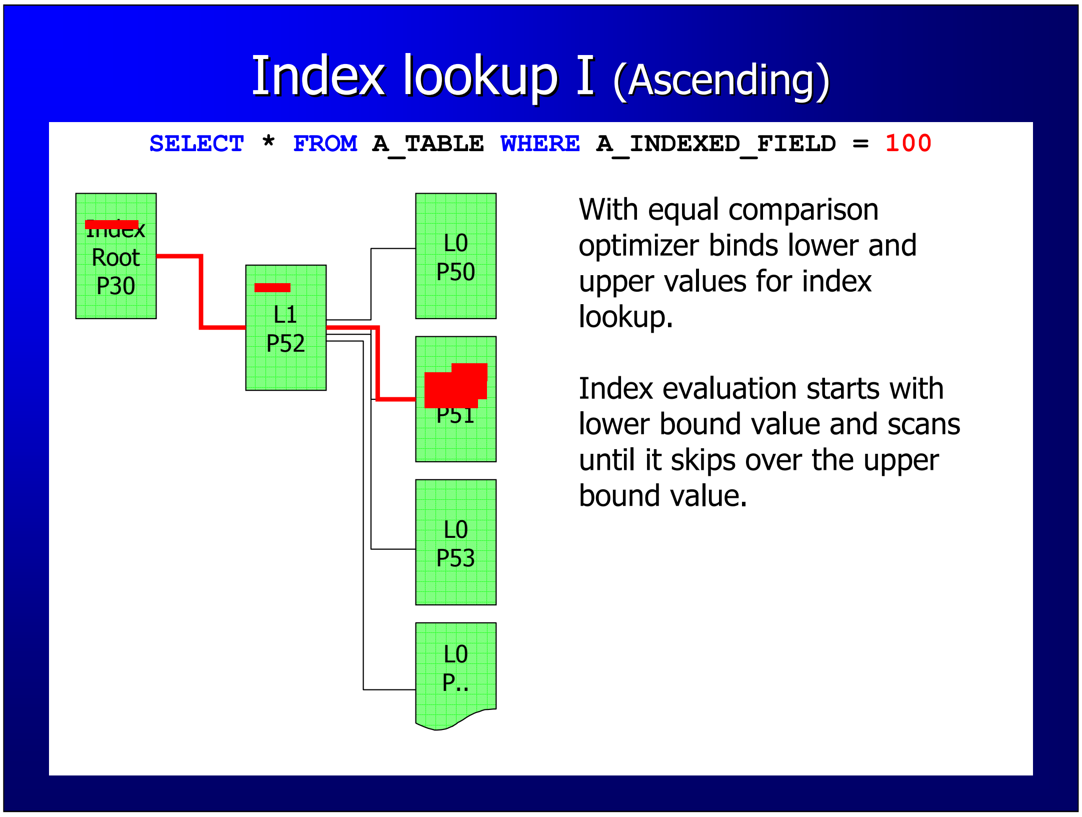
Length, prefix and number are stored compressed

Maximum index key size ¼ of page size

Record numbers are stored in non-leaf pages

40-bits record number

The index nodes in ODS 11 are stored compressed and therefore more nodes fits on 1 page, but a disadvantage was the decoding time for traversing all nodes on 1 page. To avoid this problem “jump information” has been added to the top of the page. ”Jump information” contains a few nodes with offsets and point somewhere in the page. When a node-lookup is done first is looked at the “jump information” and then the search is started in the nodes.



Index lookup I (Ascending)

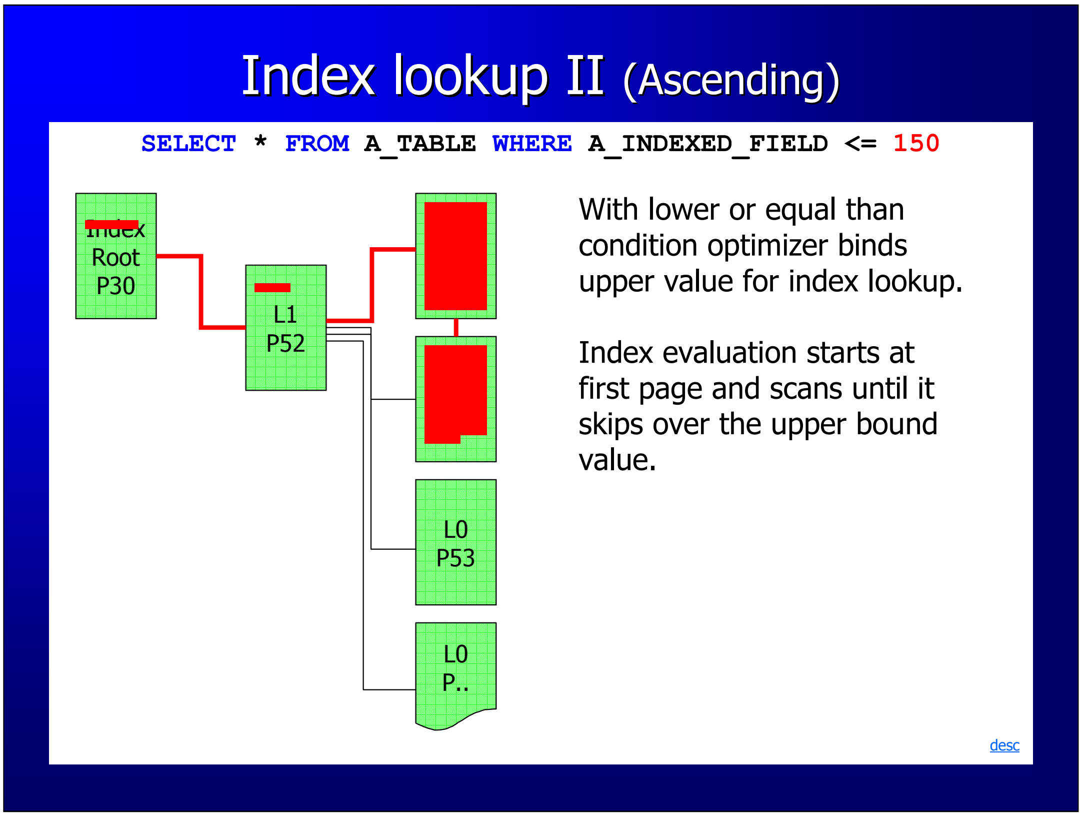
```
SELECT * FROM A_TABLE WHERE A_INDEXED_FIELD = 100
```

With equal comparison optimizer binds lower and upper values for index lookup.

Index evaluation starts with lower bound value and scans until it skips over the upper bound value.

With an index lookup it will lookup the first b-tree page in the index root page.

Next it will lookup the first matching node in the b-tree page. If this is not the leaf level page it will jump to the first next-level until the leaf page it found. From there it will start scanning until it skips over the upper bound value.



Index lookup II (Ascending)

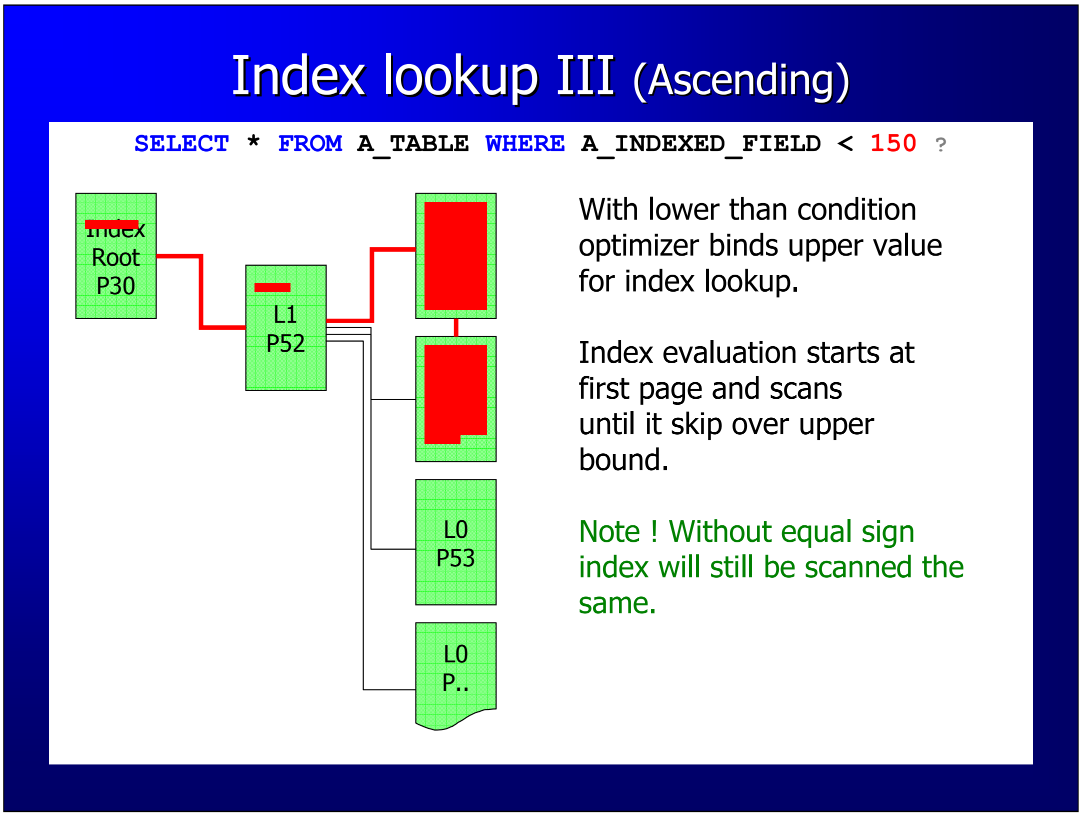
```
SELECT * FROM A_TABLE WHERE A_INDEXED_FIELD <= 150
```

With lower or equal than condition optimizer binds upper value for index lookup.

Index evaluation starts at first page and scans until it skips over the upper bound value.

desc

With lower or equal than the first leaf-level page has to be found and from there it starts scanning until it skips over the upper bound value.



Index lookup III (Ascending)

```
SELECT * FROM A_TABLE WHERE A_INDEXED_FIELD < 150 ?
```

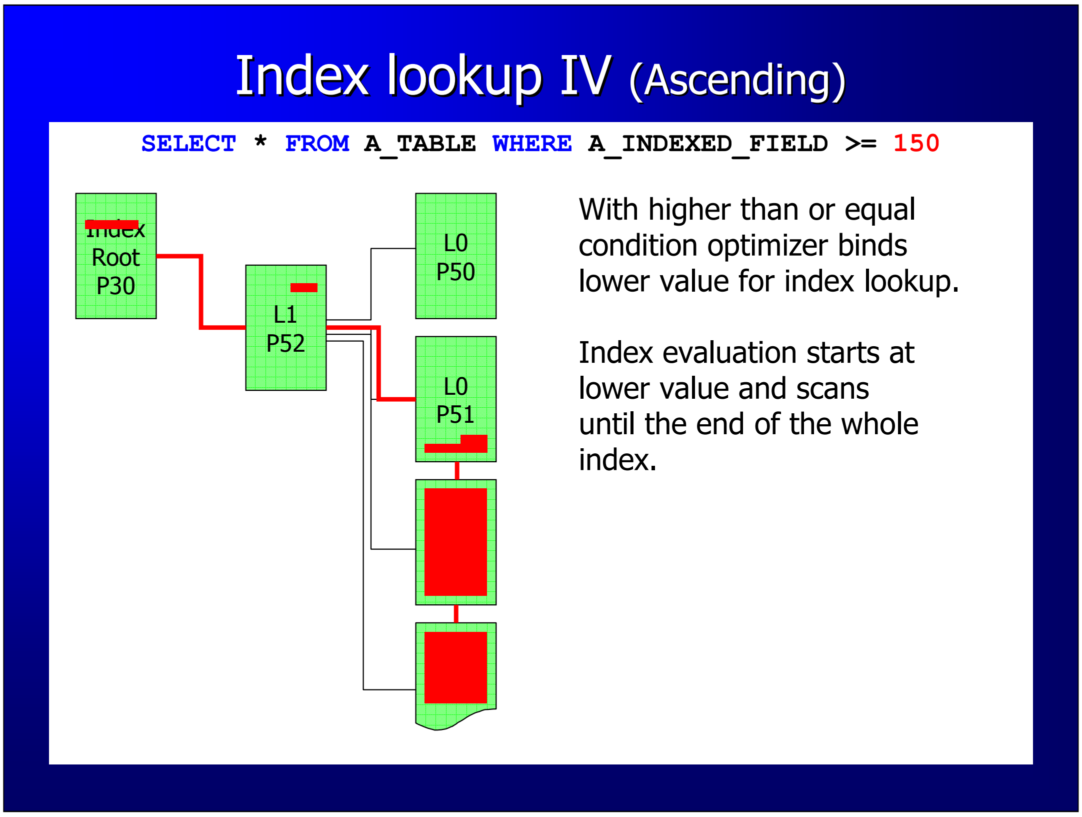
With lower than condition optimizer binds upper value for index lookup.

Index evaluation starts at first page and scans until it skip over upper bound.

Note ! Without equal sign index will still be scanned the same.

For Firebird 1.5 the bitmap returned for “lower or equal than” and “lower than” (of course compared with the same expressions on the left and right side from the operator) is exactly the same. Which resulted in unneeded record-fetches.

This problem is fixed in Firebird 2.0

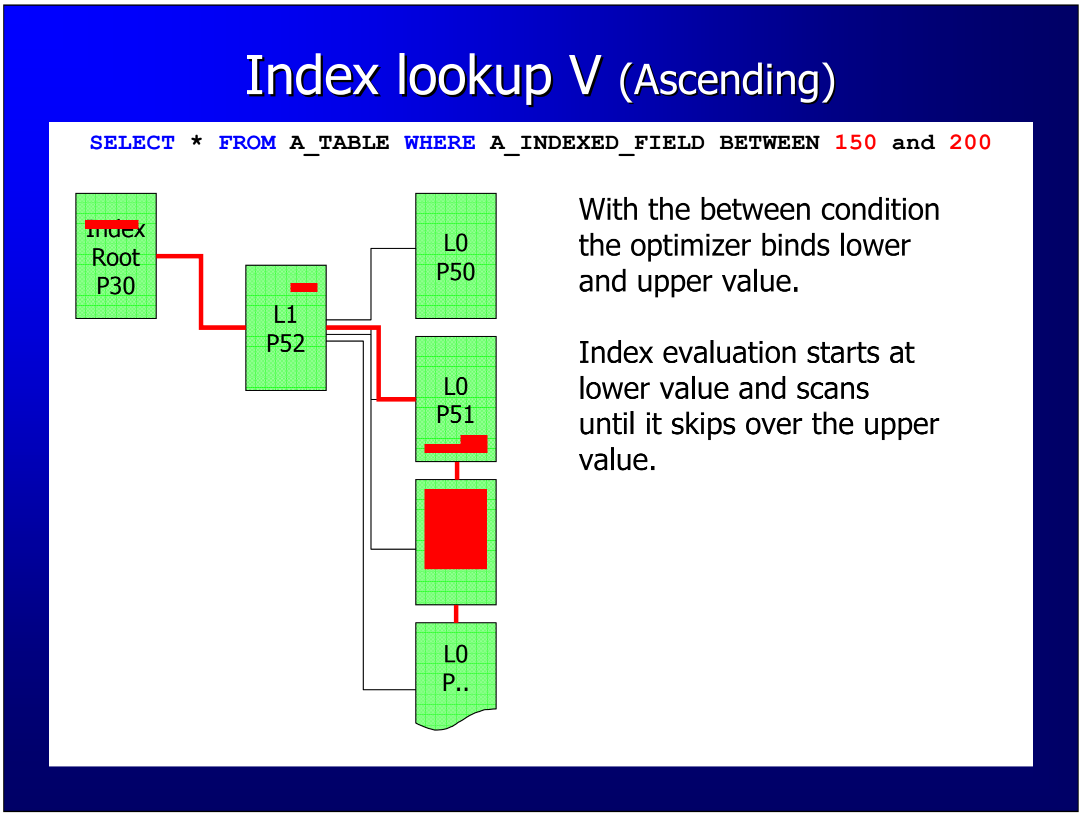


Index lookup IV (Ascending)

```
SELECT * FROM A_TABLE WHERE A_INDEXED_FIELD >= 150
```

With higher than or equal condition optimizer binds lower value for index lookup.

Index evaluation starts at lower value and scans until the end of the whole index.

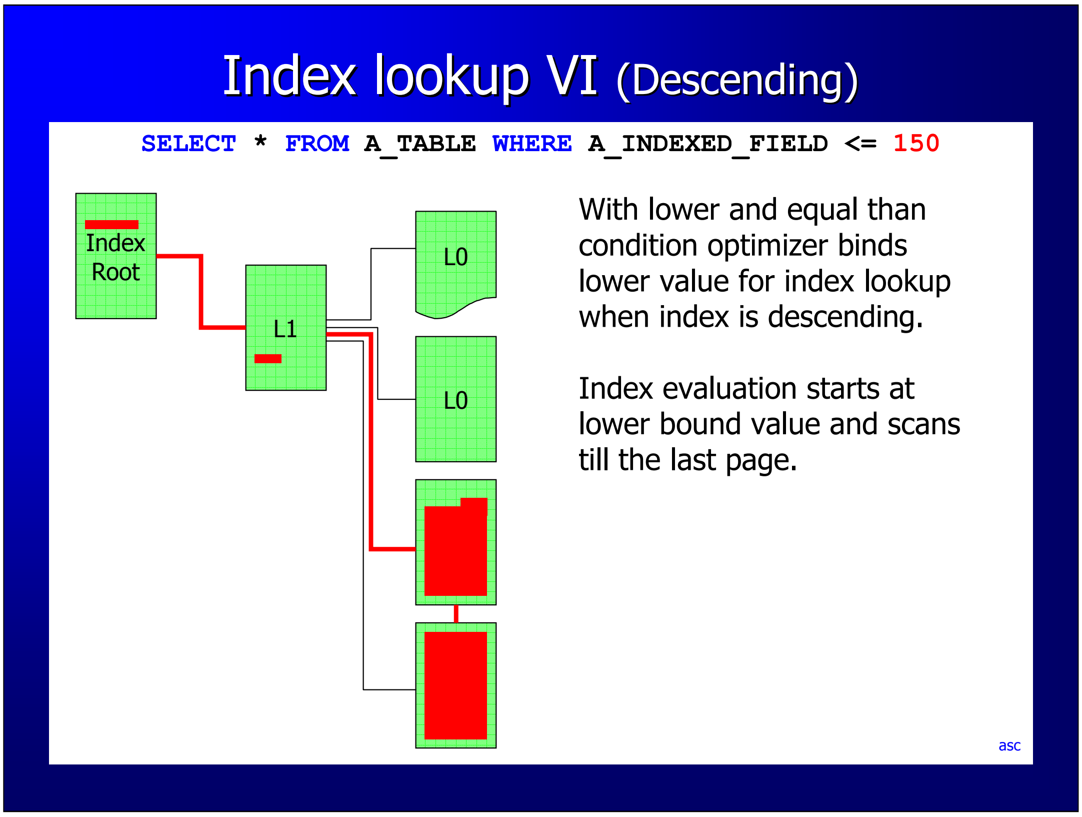


Index lookup V (Ascending)

```
SELECT * FROM A_TABLE WHERE A_INDEXED_FIELD BETWEEN 150 and 200
```

With the between condition the optimizer binds lower and upper value.

Index evaluation starts at lower value and scans until it skips over the upper value.



Index lookup VI (Descending)

```
SELECT * FROM A_TABLE WHERE A_INDEXED_FIELD <= 150
```

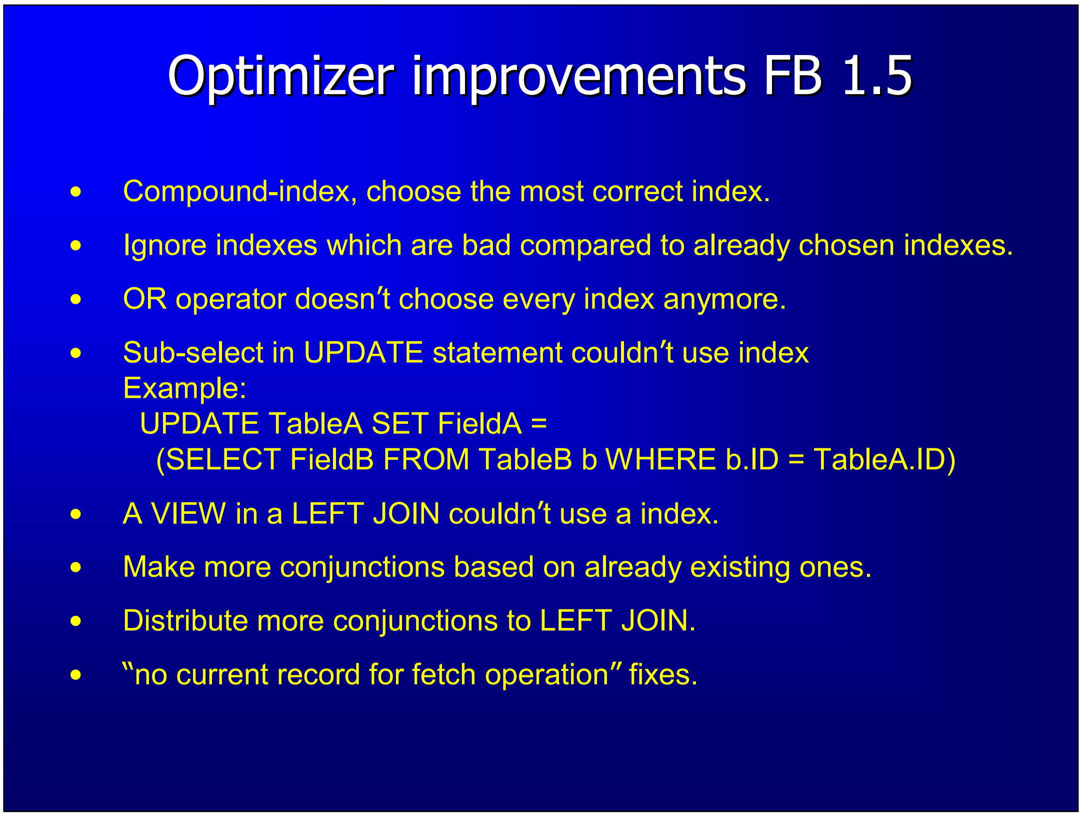
With lower and equal than condition optimizer binds lower value for index lookup when index is descending.

Index evaluation starts at lower bound value and scans till the last page.

For ODS 10 NULLS are always stored at the end and this can cause unneeded record fetches with an descending index.

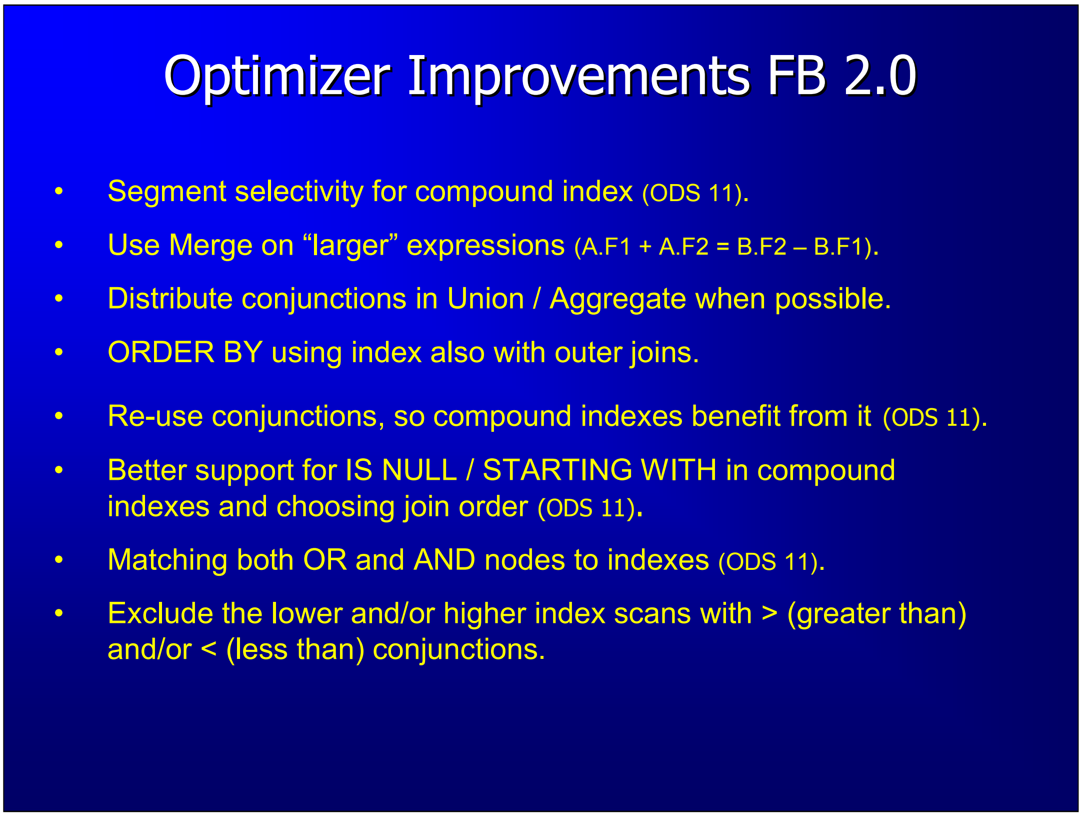
In ODS 11 NULLS are stored at the front with an ascending index and at the end by a descending index.

Prefer always an ascending index and create only a descending index when you really need them for navigation (ORDER BY), but first think twice.



Optimizer improvements FB 1.5

- Compound-index, choose the most correct index.
- Ignore indexes which are bad compared to already chosen indexes.
- OR operator doesn't choose every index anymore.
- Sub-select in UPDATE statement couldn't use index
  Example:
  ```
    UPDATE TableA SET FieldA =
      (SELECT FieldB FROM TableB b WHERE b.ID = TableA.ID)
  ```
- A VIEW in a LEFT JOIN couldn't use a index.
- Make more conjunctions based on already existing ones.
- Distribute more conjunctions to LEFT JOIN.
- "no current record for fetch operation" fixes.



Optimizer Improvements FB 2.0

- Segment selectivity for compound index (ODS 11).
- Use Merge on “larger” expressions (A.F1 + A.F2 = B.F2 – B.F1).
- Distribute conjunctions in Union / Aggregate when possible.
- ORDER BY using index also with outer joins.
- Re-use conjunctions, so compound indexes benefit from it (ODS 11).
- Better support for IS NULL / STARTING WITH in compound indexes and choosing join order (ODS 11).
- Matching both OR and AND nodes to indexes (ODS 11).
- Exclude the lower and/or higher index scans with > (greater than) and/or < (less than) conjunctions.

Future Optimizer Improvements

- OUTER Merge.
- Cached statements.
- Move LEFT JOINs to bottom when possible.
- Memory table.
- Hash JOIN.
- Histogram information.
- Out of storage order depending on FIRST/SKIP.

Questions?

Thank you for your attention