Understanding the Optimizer II

Global topics

- Introduction
- Join order
- Stream access
- Filter conditions and indexes
- Subqueries
- Questions?

Optimizer & Examples

Arno Brinkman

BISIT engineering b.v.

ABVisie

firebird@abvisie.nl

European Firebird conference 2005

Before testing queries

Be sure :

- you are testing against "real" data.
- index selectivity's are up to date.

Tools :

- Test data generator to fill a database
- PLAN analyzer

When testing statements before they go in practice you should use realistic data to avoid surprises. The best you can do is filling your tables with a lot of data (using test-data generator tools for example) which will reflect the practical use as much as possible. An production example would be even better.
Keep also the selectivity's up to date so the optimizer can make good decisions.

Use a PLAN analyzer to show you the execution path chosen by the engine. In most database tools you'll find a possibility to at least read the PLAN.

Measurements

- Prepare time

- Execution time
  When measuring execution time don't forget to fetch all records!

- Network

Measure:

-Prepare: You can keep prepare time low by caching the prepared query (thus only for the first time).

-Execution: Fetch all records you would finally fetch. Using Count(*) will fetch all records, but doesn't send much data over the wire.

-Network: Network traffic can be measured by the difference between a Count(*) call and using the fields in the select list.

Performance Analysis

- PLAN
  read from left to right

- Reads / Writes
  - Non-indexed
  - Indexed

Reads and writes gives you the number of successful record-fetches/record-updates.

SQL syntax

SQL '89

```
SELECT
  *
FROM
  RDB$RELATIONS r, RDB$RELATION_FIELDS rf ,RDB$RELATION_CONSTRAINTS rc
WHERE
  rf.RDB$RELATION_NAME = r.RDB$RELATION_NAME and
  rc.RDB$RELATION_NAME = r.RDB$RELATION_NAME
```

Prefer SQL 92 syntax above SQL 89 !

SQL '92

```
SELECT
  *
FROM
  RDB$RELATIONS r
  JOIN RDB$RELATION_FIELDS rf ON (rf.RDB$RELATION_NAME = r.RDB$RELATION_NAME)
  JOIN RDB$RELATION_CONSTRAINTS rc ON (rc.RDB$RELATION_NAME = r.RDB$RELATION_NAME)
```

SQL 89 is still a valid syntax, but prefer SQL 92 because it's much better readable for everyone.

SQL syntax

NEVER MIX SQL 92 and SQL 89 syntax!

```
SELECT
  *
FROM
  RDB$RELATIONS r,
  RDB$RELATION_FIELDS rf
  LEFT JOIN RDB$RELATION_CONSTRAINTS rc ON
     (rc.RDB$RELATION_NAME = r.RDB$RELATION_NAME)
WHERE
  rf.RDB$RELATION_NAME = r.RDB$RELATION_NAME
```

Join order I

```
SELECT * FROM
  Table_1000 t1
  JOIN Table_100 t2 ON (t2.ID = t1.ID)
  JOIN Table_10 t3 ON (t3.ID = t2.ID)
```

- T1
- T1, T2
- T1, T2, T3
- T1, T3
- T1, T3, T2
- T2
- T2, T1
- T2, T1, T3
- T2, T3
- T2, T3, T1
- T3
- T3, T1
- T3, T1, T2
- T3, T2
- T3, T2, T1

Assume that in this example all the ID's of the tables are unique. Then the optimizer (ODS 10) will try all the combinations you see and pick the cheapest cost from it.

Join order I (ODS 11)

```
SELECT * FROM
  Table_1000 t1
  JOIN Table_100 t2 ON (t2.ID = t1.ID)
  JOIN Table_10 t3 ON (t3.ID = t2.ID)
```

- T3
- T3, T2
- T3, T2, T1

- T2

- T1

When using Firebird 2.0 with ODS 11 the number of combinations is limited to chose above.

The number of combinations starting with T2 and T1 stops already by the first combination, because the cost is already higher.

Join order I

```sql
SELECT * FROM
  Table_1000 t1
  JOIN Table_100 t2 ON (t2.ID = t1.ID)
  JOIN Table_10 t3 ON (t3.ID = t2.ID)
```

Table_1000

17 data pages, format length 110, PK_TABLE_1000 (ID), selectivity 0.001

Table_100

2 data pages, format length 110, PK_TABLE_100 (ID), selectivity 0.010

Table_10

1 data page, format length 110, PK_TABLE_10 (ID), selectivity 0.100

```sql
PLAN JOIN (T3 NATURAL,T2 INDEX (PK_TABLE_100),T1 INDEX (PK_TABLE_1000))
```

The format length for the tables is the same, because the table structure is the same. Only the number of data pages is different due the number of records. Selectivity is up to date here.

Looking at the PLAN you see that the optimizer has chosen T3, T2, T1 as join order, because T3 has the least number of records it starts with that table. The total cost here is approximately 30 (10 + (10 * 1) + (10 * 1)). 10 is the cardinality for the first table. Cost for the unique fetches is always 1 and thus (10 * 1) for position T2 and T1 is added.

Statistics

```
SELECT
  i.RDB$RELATION_NAME,
  i.RDB$INDEX_NAME,
  i.RDB$STATISTICS
FROM
  RDB$INDICES i
WHERE
  i.RDB$RELATION_NAME IN ('TABLE_10', 'TABLE_100', 'TABLE_1000')
```

You can use this statement to retrieve the index selectivity values.

Statistics (ODS 11)

```
SELECT
  i.RDB$RELATION_NAME,
  i.RDB$INDEX_NAME,
  i.RDB$STATISTICS,
  ixs.RDB$FIELD_NAME,
  ixs.RDB$STATISTICS
FROM
  RDB$INDICES i
  JOIN RDB$INDEX_SEGMENTS ixs ON (ixs.RDB$INDEX_NAME = i.RDB$INDEX_NAME)
WHERE
  i.RDB$RELATION_NAME IN ('TABLE_10', 'TABLE_100', 'TABLE_1000')
ORDER BY
  i.RDB$RELATION_NAME,
  i.RDB$INDEX_NAME,
  ixs.RDB$FIELD_POSITION
```

In ODS 11 selectivity is also stored per segment. This statement can be used to get the segment selectivity's.

Join order II

```
SELECT * FROM
  Table_1000 t1
  JOIN Table_100 t2 ON (t2.ID = t1.ID)
  LEFT JOIN Table_10 t3 ON (t3.ID = t2.ID)
```

- T1
- T1, T2
- T2
- T2, T1

```
PLAN JOIN (JOIN (T2 NATURAL,T1 INDEX (PK_TABLE_1000)),T3 INDEX (PK_TABLE_10))
```

When a OUTER JOIN is used (in this example LEFT) then the no join order decision can be made between all the tables. Only the combinations for the inner joins can be made (in this example T1 and T2).
Looking at the PLAN you'll notice two JOINs where as the first JOIN is for the LEFT OUTER and the second for the INNER JOIN.

Join order III

```
SELECT * FROM
  Table_1000 t1
  LEFT JOIN Table_100 t2 ON (t2.ID = t1.ID)
  JOIN Table_10 t3 ON (t3.ID = t2.ID)
```

The optimizer can not change the order.

Enter the LEFT JOINs as low as possible.

```
PLAN JOIN (JOIN (T1 NATURAL,T2 INDEX (PK_TABLE_100)),T3 INDEX (PK_TABLE_10))
```

Using the OUTER in the middle as shown in this example the optimizer is not able to make decision about the JOIN order. When you need LEFT JOINs place them so low as possible so the optimizer can make decisions for the JOIN order where possible.

Join order IV - VIEW

```
CREATE VIEW VIEW1 (ID) AS
  SELECT t1.ID FROM
    Table_1000 t1
    JOIN Table_100 t2 ON (t2.ID = t1.ID)
    JOIN Table_10 t3 ON (t3.ID = t2.ID)
```

```
SELECT * FROM
  Table_10 t1
  JOIN View1 v1 ON (v1.ID = t1.ID)
```

When possible INNER JOINs are combined to one INNER JOIN.

```
PLAN JOIN (T1 NATURAL,V1 T3 INDEX (PK_TABLE_10),V1 T2 INDEX (PK_TABLE_100)
     ,V1 T1 INDEX (PK_TABLE_1000))
```

In a phase before calling the optimizer multiple INNER JOINs are combined together to 1 INNER JOIN where possible. VIEWs are flattened where possible and combined too.

Join order V - VIEW

```
CREATE VIEW VIEW2 (ID) AS
  SELECT t1.ID FROM
    Table_1000 t1
    JOIN Table_100 t2 ON (t2.ID = t1.ID)
    LEFT JOIN Table_10 t3 ON (t3.ID = t2.ID)
```

```
SELECT * FROM
  Table_10 t1
  JOIN View2 v1 ON (v1.ID = t1.ID)
```

Can not combine the INNER JOINs due the LEFT JOIN at the end.

```
PLAN JOIN(JOIN(JOIN(V1 T2 NATURAL,V1 T1 INDEX (PK_TABLE_1000),
    V1 T3 INDEX (PK_TABLE_10),T1 INDEX (PK_TABLE_10))
```

When a VIEW contains an OUTER JOIN at the end it isn't possible to combine the tables together to 1 INNER JOIN.

As for this example it isn't possible to put the tables T1 and T1, T2 together into 1 INNER JOIN, but of course a join order decision between t1 and t2 inside the VIEW can still be made.

Stream access I

```
CREATE TABLE Relations (
  RelationID  INTEGER  NOT NULL,
  FirstName   VARCHAR(35),
  LastName    VARCHAR(35),
  CONSTRAINT PK_Relations PRIMARY KEY (RelationID)
)
```

```
SELECT
  Count(*)
FROM
  Relations r
```

```
PLAN (R NATURAL)
```

Running this select statement will cause Firebird to read the whole table. It reads all records in storage order (the order it's stored on disk) and this is the fastest way, because when it was using an index it still has to look for every record in the database. This is needed, because it has to check if the index entry was valid for our current transaction.

Stream access II

```
CREATE TABLE Relations (
  RelationID  INTEGER  NOT NULL,
  FirstName   VARCHAR(35),
  LastName    VARCHAR(35),
  CONSTRAINT PK_Relations PRIMARY KEY (RelationID)
)
```

```
SELECT
  r.*
FROM
  Relations r
ORDER BY
  r.RelationID
```

```
PLAN (R ORDER PK_RELATIONS)
```

Using the PK in the ORDER BY here causes that the optimizer chooses for an navigation through index. This will also read the whole table, but in the PK order. The data isn't stored in any specific order in the data pages, thus this is a good candidate for random disk read.

Note that I use "r.*" in this example, but normally you wouldn't do that. Only put the fields you need in the select list to keep the network traffic so small as possible.

Stream access III

The index used for an ORDER is only useful when you fetch the first x records.

```
SELECT
  FIRST 10
  r.*
FROM
  Relations r
ORDER BY
  r.RelationID
```

```
PLAN (R ORDER PK_RELATIONS)
```

With FIRST x (or just fetching the first x records at the client) it’s useful when an index is used for reading data from the disk. Now only the first 10 records are fetched from disk. Without an index on field relationID the whole table would need to be read first and sorted afterwards. Which would be more expensive for this statement.

Stream access IV

```
CREATE ASC INDEX IDX_RELATIONS_LASTNAME ON Relations(LastName)
```

```
SELECT
  r.LastName,
  Count(*)
FROM
  Relations r
GROUP BY
  r.LastName
```

```
PLAN (R ORDER IDX_RELATIONS_LASTNAME)
```

If the fields in the GROUP BY can be matched against an index (in the same order) it will use this index for navigation. Note that for this example using a SORT afterwards is faster, because all records are read. Maximum 1 index can be used for navigation.

Stream access V

How to make sure the index won't be used for navigation?

SELECT
  r.LastName,
  Count(*)
FROM
  Rrelations r
GROUP BY
  r.LastName, r.LastName
	SELECT
	  r.LastName || '',
	  Count(*)
	FROM
	  Relations r
	GROUP BY
	  1

```
PLAN SORT ((R NATURAL))
```

To avoid that an index is chosen for navigation you can add another (or the same) field to the GROUP BY clause or “concatenate with an empty string” / “add a zero”. The same can be done for the ORDER BY clause.

Stream access VI

Is it possible to use an index for a filter, while using an index for navigational access?

```
SELECT
  r.LastName,
  Count(*)
FROM
  Relations r
WHERE
  r.RelationID = 10121
GROUP BY
  r.LastName
```

```
PLAN (R ORDER IDX_RELATIONS_LASTNAME INDEX (PK_RELATIONS))
```

Firebird 1.5 will show you only PLAN (R ORDER IDX_RELATIONS_LASTNAME), but internally it will use the index available with RelationID in it. Firebird 2.0 will output the PLAN with ORDER and INDEX together.

DSQL conversions I

```sql
SELECT * FROM
  Relations r
WHERE
  r.RelationID IN (10, 11, 12)
```

⬇

```sql
SELECT * FROM
  Relations r
WHERE
  r.RelationID = 10 OR
  r.RelationID = 11 OR
  r.RelationID = 12
```

The IN predicate is converted inside the DSQL (Dynamic SQL) to multiple OR statements.

DSQL conversions II

```sql
SELECT * FROM
  Relations r
WHERE
  r.RelationID IN (SELECT rc.RelationID FROM RelationCategories rc)
```

```sql
SELECT * FROM
  Relations r
WHERE
  EXISTS(SELECT rc.RelationID FROM RelationCategories rc
    WHERE rc.RelationID = r.RelationID)
```

IN predicate with sub-query is internally converted to another form which is the same as using an EXISTS. Often is complained why the statement at the top can't use an index on the field relationID. Looking at the converted statement below you probably already understand why it isn't possible. As you'll see an index will be used for the sub-query (due the rc.RelationID = r.RelationID), but the other table will be fetched completely.

DSQL conversions III

```
SELECT * FROM
  Relations r
WHERE
  r.RelationID IN (SELECT Max(rc.RelationID) FROM RelationCategories rc)
```

```
SELECT * FROM
  Relations r
WHERE
  EXISTS(SELECT Max(rc.RelationID) FROM RelationCategories rc
    HAVING Max(rc.RelationID) = r.RelationID)
```

When the sub-query is an aggregate query (it has GROUP BY/HAVING or aggregate function in the statement) the condition goes the HAVING clause instead off the WHERE clause. This can have a big impact on performance, because the HAVING clause is performed after the grouping.

The optimizer in Firebird 2.0 will try to distribute the condition to WHERE clause, so an index can be used when available.

DSQL conversions IV

```
SELECT * FROM
  Relations r
WHERE
  r.RelationID IN (SELECT FIRST 1 rc.RelationID FROM RelationCategories rc)
```

```
SELECT * FROM
  Relations r
WHERE
  EXISTS(SELECT FIRST 1 rc.RelationID FROM RelationCategories rc
    WHERE rc.RelationID = r.RelationID)
```

Warning 2, In Firebird 1.5 and before the IN predicate would probably not do what you expect. The sub-query is evaluated for record from the master query and thus this will return all records for Relations.

Firebird 2.0 has fixed this by problem, but note that no index will be used for the sub-query. You can work-around this by using the sub-query as derived table.

```
SELECT
  *
FROM
  Relations r
  JOIN (SELECT FIRST 1 rc.RelationID FROM RelationCategories rc) rc
    ON (rc.RelationID = r.RelationID)
```

DSQL conversions V

```
SELECT * FROM
  Relations r
WHERE
  NOT r.RelationID <> 10
```

```
SELECT * FROM
  Relations r
WHERE
  r.RelationID = 10
```

In Firebird 2.0 NOT conditions are simplified where possible, so that the optimizer eventually can match it against an index.

Operators / predicates I

Can use an index:

- equals (a = b)
- less than (a < b)
- greater than (a > b)
- less than or equal (a <= b)
- greater than or equal (a >= b)
- IS NULL
- STARTING WITH
- IN with list of constants
- BETWEEN

Can't use an index:

- not equal to (a <> b)
- IS NOT NULL
- CONTAINING
- LIKE
- IN (subquery)
- NOT

Operators / predicates II

An index can only be used inside the subquery:

- IN (subquery)
- EXISTS (subquery)
- SINGULAR (subquery)
- ALL (a # ALL (subquery))
- ANY (a # ANY (subquery))
- SOME (a # SOME (subquery))

= (=, <, >, <=, >=, <>)

Filter conditions - LIKE

The optimizer add an STARTING WITH if the LIKE doesn't start with an pattern-matching character.

```
CREATE ASC INDEX IDX_RELATIONS_LASTNAME ON Relations(LastName)
```

```
SELECT
  r.*
FROM
  Relations r
WHERE
  r.LastName LIKE 'Sp%'
```

```
                    internal
SELECT
  r.*
FROM
  Relations r
WHERE
  r.LastName LIKE 'Sp%' and
  r.LastName STARTING WITH 'Sp'
```

```
PLAN (R INDEX (IDX_RELATIONS_LASTNAME))
```

Note that when you use a parameter never a STARTING WITH is added, because the value of the parameter can contain anything.

Filter conditions - BETWEEN

The optimizer converts a “between” conjunction into a “greater than or equal” and a “less than or equal” conjunction.

```
SELECT
  r.*
FROM
  Relations r
WHERE
  r.LastName BETWEEN 'D' and 'A'
```

internal

```
SELECT
  r.*
FROM
  Relations r
WHERE
  r.LastName >= 'D' and
  r.LastName <= 'A'
```

```
PLAN (R INDEX (IDX_RELATIONS_LASTNAME))
```

Filter conditions - OR

```
SELECT
  r.*
FROM
  Relations r
WHERE
  r.RelationID = 11 or
  r.RelationID = 23 or
  r.RelationID = 44 or
  r.RelationID = 56
```

```
PLAN (R INDEX (PK_RELATIONS, PK_RELATIONS, PK_RELATIONS, PK_RELATIONS))
```

With an OR filter only indexes can be used when every condition can use a index, because if one condition can't use an index it has to be evaluated against every record in the table.

Filter conditions – AND

```
SELECT
  r.*
FROM
  Relations r
WHERE
  r.RelationID => 1000 and
  r.LastName < 'F'
```

```
PLAN (R INDEX (PK_RELATIONS,IDX_RELATIONS_LASTNAME))
```

If possible and it's interesting (versus cost) two indexes will be used and internally the indexed results are AND-ed. The final result (list of record numbers) is used to lookup the records.

Filter conditions – compound index

Field order in compound index is important!

```
CREATE ASC INDEX IDX_RELATIONS_FIRSTNAME ON Relations(FirstName)
CREATE ASC INDEX IDX_RELATIONS_FIRSTLASTNAME ON Relations(FirstName, LastName)
```

```
SELECT
  r.*
FROM
  Relations r
WHERE
  r.FirstName = 'Adam' and
  r.LastName = 'Kern'
```

```
PLAN (R INDEX (IDX_RELATIONS_FIRSTLASTNAME))
```

An index with two or more fields (compound index) can be very useful if you filter a lot against the same fields with an equality operator. This index will probably also have a good selectivity, because mostly the number if distinct nodes will decrease. Using a field which is unique in a compound index meant in most cases that the index is in fact unneeded/wrong.

Filter conditions – compound index

When the first segments are not matched with an equal's operator then the next segments cannot be efficiently used.

```
CREATE ASC INDEX IDX_RELATIONS_FIRSTLASTNAME ON Relations(FirstName, LastName)
```

```
SELECT
  r.*
FROM
  Relations r
WHERE
  r.FirstName > 'Adam' and
  r.LastName = 'Kern'
```

```
PLAN (R INDEX (IDX_RELATIONS_LASTNAME,IDX_RELATIONS_FIRSTNAME))
```

Filter conditions – compound index

Three indexes are used, while two compound indexes could be used?

```
SELECT r.* FROM
  Relations r
WHERE
  r.FirstName IN ('Adam', 'Jane') and
  r.LastName = 'Kern'
PLAN (R INDEX (IDX_RELATIONS_LASTNAME,IDX_RELATIONS_FIRSTNAME,
    IDX_RELATIONS_FIRSTNAME))
```

Rewrite the query:

```
SELECT r.* FROM
  Relations r
WHERE
  (r.FirstName = 'Adam' and r.LastName = 'Kern') or
  (r.FirstName = 'Jane' and r.LastName = 'Kern')
PLAN (R INDEX (IDX_RELATIONS_FIRSTLASTNAME,IDX_RELATIONS_FIRSTLASTNAME))
```

The first statement is not able to use the compound index on FirstName and LastName twice. For Firebird 1.5 you've to rewrite the query to something like the lower statement. In Firebird 2.0 the first statement will be able to use the compound index and no rewrite is needed.

The same counts when the IN predicate in the first statement were 2 OR conditions.

Filter conditions – ignore index

The optimizer can only use a index when at the left or right side from the operator 1 field is present.

Add 0 or empty string:

```
SELECT
  r.*
FROM
  Relations r
WHERE
  r.RelationID + 0 > 100 and
  r.LastName || '' = 'Kern'
```

```
PLAN (R NATURAL)
```

Add OR:

```
SELECT
  r.*
FROM
  Relations r
WHERE
  r.RelationID > 100 and
  (r.LastName = 'Kern' or 1 = 0)
```

```
PLAN (R NATURAL)
```

Filter conditions – LEFT JOIN

```
SELECT
  *
FROM
  Table_1000 t1
  LEFT JOIN Table_100 t2 ON (t2.ID = t1.ID)
  JOIN Table_10 t3 ON (t3.ID = t1.ID)
WHERE
  t2.SomeField = 'Firebird'
```

Using a filter on a LEFT JOIN let the LEFT JOIN behave as an INNER JOIN (except checking for NULL state).

```
PLAN JOIN (JOIN (T1 NATURAL,T2 INDEX (PK_TABLE_100)),T3 INDEX (PK_TABLE_10))
```

When filtering on an OUTER JOIN in the WHERE clause you let the OUTER clause behave as an INNER JOIN. The only exception here is if you’re checking for NULL states in the WHERE clause (such as for “t2.SomeField IS NULL” or “COALESCE(t2.SomeField, 0) = 0”).

This is also a way to force the order in which the tables are JOINed, but it’s recommended to let the optimizer decide. Assuming data grows and the tables are changing in size compared to each other.

Filter conditions – aggregate

Use the WHERE clause whenever possible.

```
SELECT
  r.FirstName,
  Count(*)
FROM
  Relations r
GROUP BY
  r.FirstName
HAVING
  r.FirstName = 'Jim'
```

```
SELECT
  r.FirstName,
  Count(*)
FROM
  Relations r
WHERE
  r.FirstName = 'Jim'
GROUP BY
  r.FirstName
```

```
PLAN (R ORDER IDX_RELATIONS_FIRSTLASTNAME)
```

Filters in the HAVING clause cannot use an index. Always prefer filters in the WHERE clause, only filters on aggregate functions should be put in the HAVING clause.

Firebird 2.0 will try to distribute the HAVING clause to the WHERE clause by itself.

Aggregate

Max/Min Will use an index when possible.

```
SELECT Min(FirstName) FROM Relations r
```

```
PLAN (R ORDER IDX_RELATIONS_FIRSTNAME_ASC)
```

```
SELECT Max(FirstName) FROM Relations r
```

```
PLAN (R ORDER IDX_RELATIONS_FIRSTNAME_DESC)
```

```
SELECT Max(FirstName), Count(*) FROM Relations r
```

```
PLAN (R NATURAL)
```

When a single Min or Max aggregate function is used it will try to use an index for navigation. Min can only use Ascending indexes and Max can only use descending indexes.

Subqueries I

- Have their own PLAN.
- Correlated subqueries are executed for every 'row'.

```
SELECT
  r.LastName || ', ' || r.FirstName,
  (SELECT c.Description FROM Categories c WHERE c.CategoryID = rc.CategoryID)
FROM
  RelationCategories rc
  JOIN Relations r ON (r.RelationID = rc.RelationID)
ORDER BY
  1
```

```
PLAN (C INDEX (PK_CATEGORIES))
PLAN SORT(JOIN(R NATURAL,RC INDEX (FK_RELCAT_RELATIONS)))
```

Subqueries II

Note! When ORDER BY or GROUP BY clause refer to a subquery in select list then this subquery will be executed twice.

```
SELECT
  r.LastName || ', ' || r.FirstName,
  (SELECT c.Description FROM Categories c WHERE c.CategoryID = rc.CategoryID)
FROM
  RelationCategories rc
  JOIN Relations r ON (r.RelationID = rc.RelationID)
ORDER BY
  2
```

```
PLAN (C INDEX (PK_CATEGORIES))
PLAN (C INDEX (PK_CATEGORIES))
PLAN SORT(JOIN(R NATURAL,RC INDEX (FK_RELCAT_RELATIONS)))
```

UNION

- Every query-item has a PLAN.
- Prefer UNION ALL above UNION.

```
SELECT r.LastName || ', ' || r.FirstName,
FROM Relations r
WHERE r.RelationID >= 1 and r.RelationID <= 10
UNION ALL
SELECT r.LastName || ', ' || r.FirstName,
FROM Relations r
WHERE r.RelationID >= 1 and r.RelationID <= 10
ORDER BY
  2
```

```
PLAN (R INDEX (PK_RELATIONS))
PLAN (R INDEX (PK_RELATIONS))
```

Every query used on the left and right side of an UNION has it's own PLAN. When you don't need to eliminate duplicate rows use the UNION ALL, because this doesn't use the distinct operation afterwards. Using just UNION will always cause an distinct being added internally, but you can't read this info from the PLAN output.

UNION - VIEW

UNION is processed first.

```
CREATE VIEW View3 (RelationID) AS
 SELECT rc.RelationID
 FROM RelationCategories rc
 WHERE rc.CategoryID = 5
 UNION ALL
 SELECT rc.RelationID
 FROM RelationCategories rc
 WHERE rc.CategoryID = 8
```

```
SELECT
  r.*
FROM
  Relations r
  JOIN View3 v ON (v.RelationID = r.RelationID)
```

```
PLAN JOIN((V RC INDEX (FK_RELCAT_CATEGORIES))
  PLAN (V RC INDEX (FK_RELCAT_CATEGORIES)),R INDEX (PK_RELATIONS))
```

Unions are processed first on the same level. That's why you see here the view at the beginning of the PLAN.

UNION - distribute

```
SELECT
  v.*
FROM
  View3 v
WHERE
  v.RelationID BETWEEN 4000 and 5000
```

Firebird 1.5

```
PLAN (V RC INDEX (FK_RELCAT_CATEGORIES))
PLAN (V RC INDEX (FK_RELCAT_CATEGORIES))
```

Firebird 2.0

```
PLAN (V RC INDEX (PK_RELATIONCATEGORIES))
PLAN (V RC INDEX (PK_RELATIONCATEGORIES))
```

For Firebird 1.5 the filter in the WHERE clause will only be evaluated after the whole VIEW is executed, while the relationID is also part of an index. In Firebird 2.0 the WHERE clause on a UNION (in this case the VIEW, but it could also be a derived table) will be distributed and other indexes could probably be chosen. Such as in this example where the index from the primary key could be used.

Selectable Stored Procedures

Selectable stored procedures are processed first.

```
SELECT
  *
FROM
  Relations r
  LEFT JOIN StoredProcedure(r.RelationID) ON (1 = 1)
```

Note that stored procedures are cached (prepared statements) inside the engine. When you add an index which you expect to be used by the stored procedure, but you already run the procedure once. This index will not be used, because it doesn't come in the optimizer anymore.

DON'T USE EXPLICIT PLANs!

DON'T
USE
EXPLICIT
PLANs!

Questions?

The End