Firebird 3.0 statistics and plans

Dmitry Kuzmenko, IBSurgeon

Firebird 2017 Tour: Performance Optimization

- Firebird Tour 2017 is organized by Firebird Project, IBSurgeon and IBPhoenix, and devoted to Firebird Performance.
- The Platinum sponsor is Moscow Exchange
- Tour's locations and dates:
  - October 3, 2017 – Prague, Czech Republic
  - October 5, 2017 – Bad Sassendorf, Germany
  - November 3, 2017 – Moscow, Russia

MOSCOW
EXCHANGE

- Platinum Sponsor
- Sponsor of
  - «Firebird 2.5 SQL Language Reference»
  - «Firebird 3.0 SQL Language Reference»
  - «Firebird 3.0 Developer Guide»
  - «Firebird 3.0 Operations Guide»
- Sponsor of Firebird 2017 Tour seminars
- www.moex.com

Firebird Foundation
SPONSOR
Supporting
Firebird development

PLATINUM

IBSurgeon

HQBird
Advanced Firebird for Big Databases

- Replication, Recovery and Optimization for Firebird and InterBase since 2002
- Platinum Sponsor of Firebird Foundation
- Based in Moscow, Russia

www.ib-aid.com

Agenda

- New elements for table statistics
  - Including blob information
- New elements for index statistics

- Plan elements
- Explained plans
- Optimizer enhancements
  - Firebird 3 and 4

NEW STATISTICS ELEMENTS

How to get statistics

- Gstat –r
- Gstat –r –t tablename1 –t tablename2…
- Services API
- HQbird Database Analyst

Tables

- JOB (129)
    - Primary pointer page: 228, Index root page: 229
    - **Total formats: 1, used formats: 1**
    - Average record length: 65.58, total records: 31
    - Average version length: 0.00, total versions: 0, max versions: 0
    - **Average fragment length: 0.00, total fragments: 0, max fragments: 0**
    - **Average unpacked length: 96.00, compression ratio: 1.46**
    - **Pointer pages: 1, data page slots: 3**
    - Data pages: 3, average fill: 72%
    - **Primary pages: 1, secondary pages: 2, swept pages: 1**
    - **Empty pages: 0, full pages: 1**
    - **Blobs: 39, total length: 4840, blob pages: 0**
        - **Level 0: 39, Level 1: 0, Level 2: 0**
    - Fill distribution:
        - 0 - 19% = 0
        - 20 - 39% = 0
        - 40 - 59% = 1
        - 60 - 79% = 1
        - 80 - 99% = 1

Total formats: 1, used formats: 1

- Number of table structure changes (except triggers and indices)
- Limited to 256
- After limit exceeded, you need to do backup/restore

- Used formats – how many formats used by primary records. Number of all used formats is unknown (less or equal of Formats)

Primary, Secondary – new storage concept

- Primary
  - Primary record versions - insert
  - Backversions
- Secondary
  - Backversions, record fragments - update/delete
  - Small blobs (level 0)

Swept

- Processed by garbage collector or sweep
  - Sweep skips swept pages
- Used for primary pages
- When no work for garbage collector
- Cleared when new version is created on the data page

Average fragment length: 0.00, total fragments: 0, max fragments: 0

- Fragments - records that does not fit at a page
  - Big records
  - Big record+versions chain
- Max fragments – the most number of fragments for some record

Packing

- **Average unpacked length: 96.00, compression ratio: 1.46**
- Average record length: 65.58

- 96 / 65.58 = 1.46

Empty, full

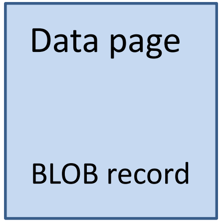
- Full – when there is no space to place new record (version)
- Empty – empty, while not gathered into 8-pages extent. These pages are marked as unused only when all pages in extent are empty.

Blobs, blob levels

- **Blobs: 463, total length: 248371310, blob pages: 15410**
  Level 0: 0, Level 1: 463, Level 2: 0

- **Level 0** – fits to the data page. Record data is sparsed.
  - Page of 4096 bytes can hold blob of 4052 bytes
- **Level 1** – pointers to the blob pages.
  - Blobs bigger than page size, and up to ~4mb size can rarefact data same way as 4052 blobs. Because 4052 bytes can fit 1013 links to the blob pages
- **Level 2** – pointer to the blob pointer page, that contain pointers to the blob pages

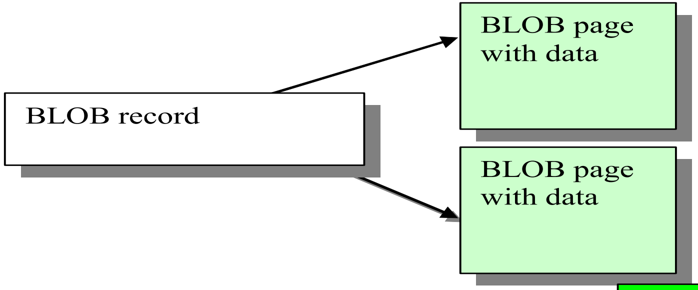
BLOB Levels



Data page

BLOB record

- Level 0 – at data page, as record data

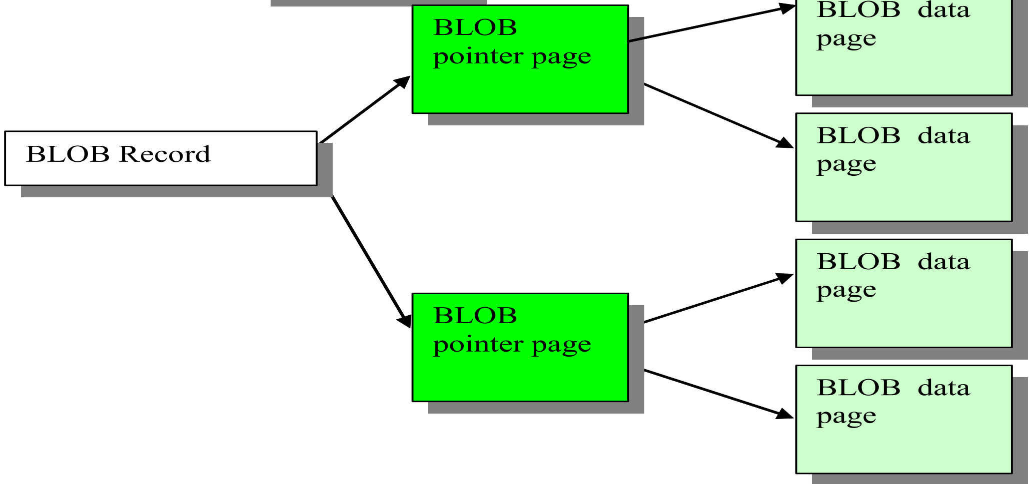


BLOB record

BLOB page with data

BLOB page with data

- Level 1 – pointers to the blob pages BLOB



BLOB Record

BLOB pointer page

BLOB pointer page

BLOB data page

BLOB data page

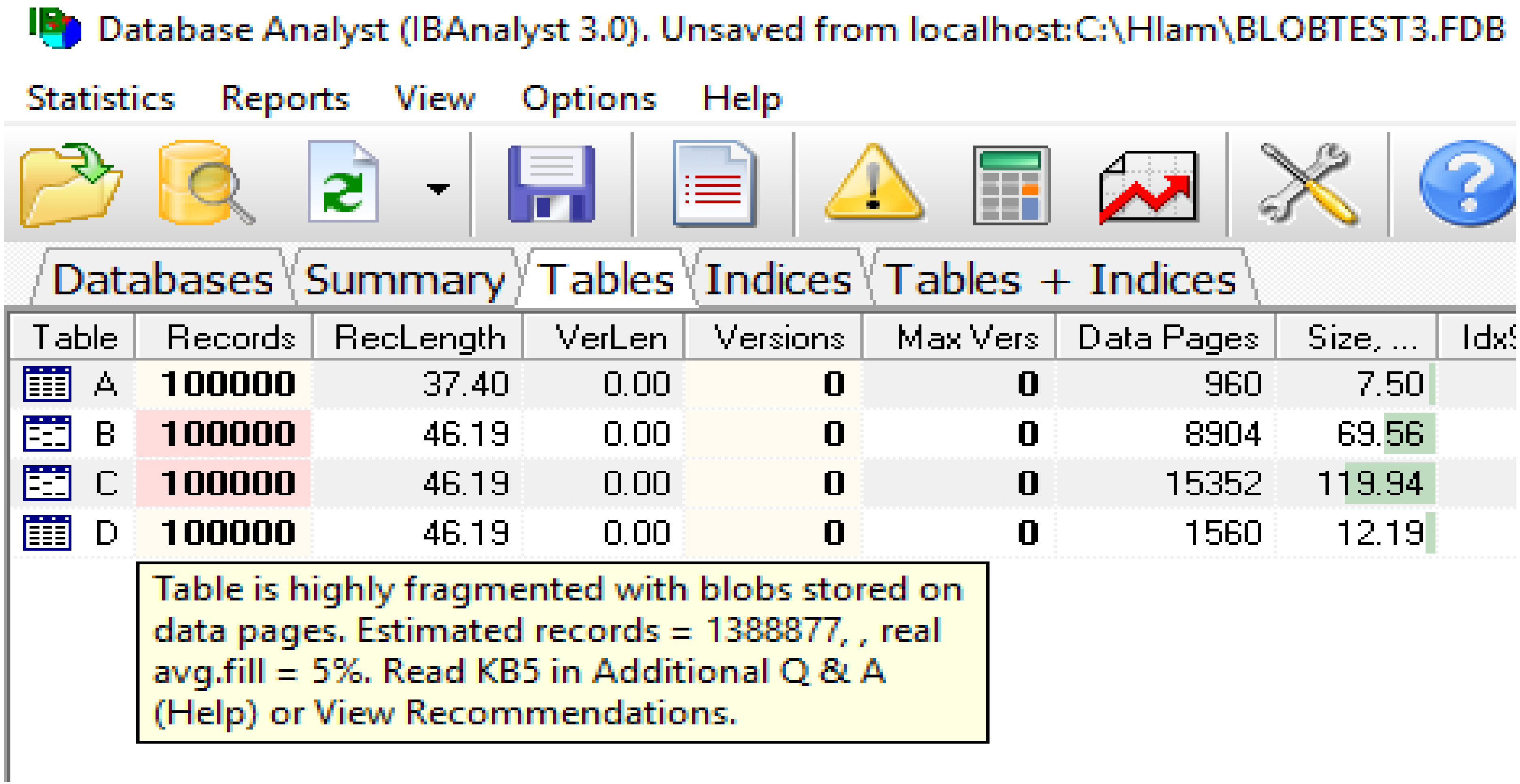
BLOB data page

BLOB data page

- Level 2 – pointers to pointers (BLOB pages with pointers)

What was earlier (ODS < 12)

- Small blobs (level 0) sparse record data, because they fit at data page
- Records could be highly sparsed, causing performance loss on scans without accessing blobs
- To avoid this small blobs needed to be moved to separate table, linked 1:1 to the main table



Database Analyst (IBAnalyst 3.0). Unsaved from localhost:C:\Hlam\BLOBTEST3.FDB

Table	Records	RecLength	VerLen	Versions	Max Vers	Data Pages	Size, ...	Idx
A	**100000**	37.40	0.00	**0**	**0**	960	7.50	
B	**100000**	46.19	0.00	**0**	**0**	8904	69.56	
C	**100000**	46.19	0.00	**0**	**0**	15352	119.94	
D	**100000**	46.19	0.00	**0**	**0**	1560	12.19	

Table is highly fragmented with blobs stored on data pages. Estimated records = 1388877, , real avg.fill = 5%. Read KB5 in Additional Q & A (Help) or View Recommendations.

Blob level 0

- Now “blob record”, i.e. blob contents, stored at a secondary page, while record is at primary page
- Eliminates data page sparse
- Makes scan operations much faster
  - Anything that does not touch blob data

Blob level 1

- **Blobs: 463, total length: 248371310, blob pages: 15410**
  Level 0: 0, Level 1: 463, Level 2: 0

- Here all blobs
  - Bigger than page size (16k) (no Level 0)
  - Less than 64mb (no level 2)
  - Do not interleave record data

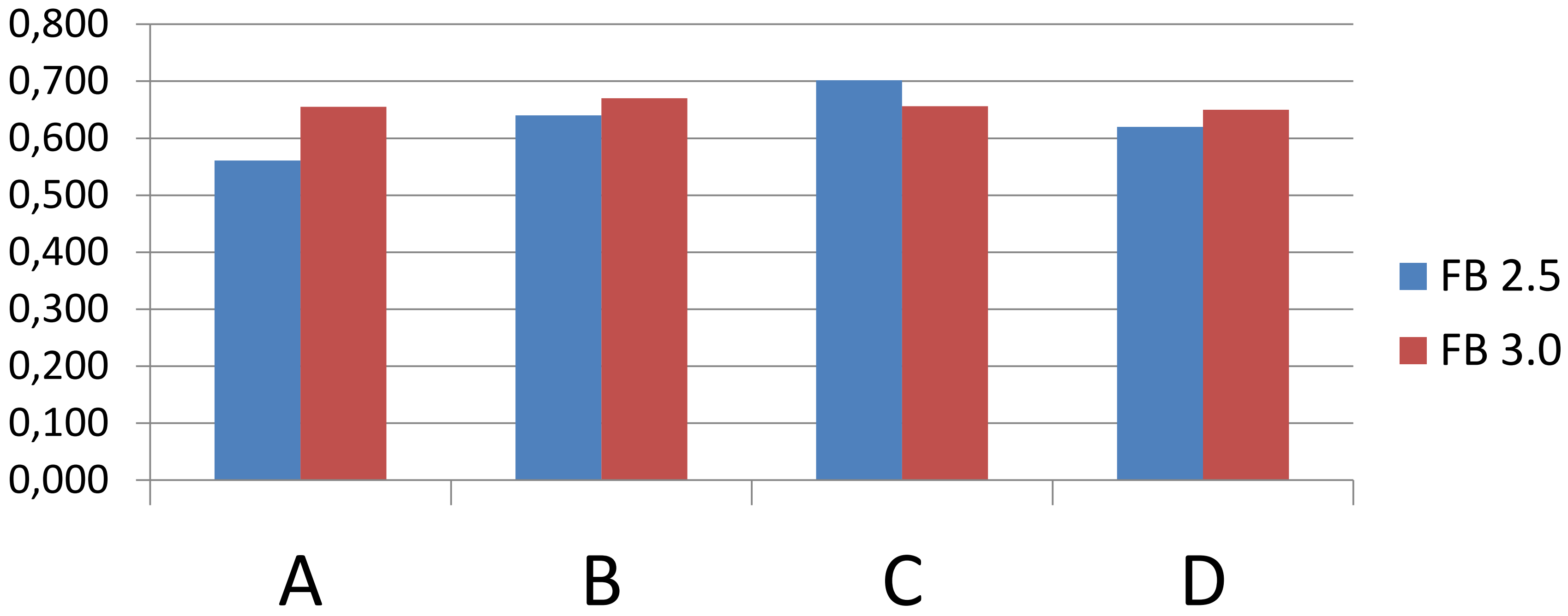
Blob level 2

- Blob record points to pages that contain pointers to blob pages
- For 16k page size blob size must be bigger than 64mb to get Level 2

Test

- Page size = 8192
- Tables, 100k records, random data
  - A – no blobs at all
  - B – random blobs from 128 to 1024 bytes size
    - Fits at data page – level 0
  - C – fixed blobs 1024 bytes size
    - Fits at data page – level 0
  - D – fixed blobs 9000 bytes size
    - Goes to separate blob page – level 1
- Reading all fields except blob
  - Fetch all, select count

Reading speed, ms



Performance with B and C may decrease up to 22%

SELECT * FROM C

Fetch All

PLAN (C NATURAL)

------ Performance info ------

Prepare time = 16ms

Execute time = 875ms

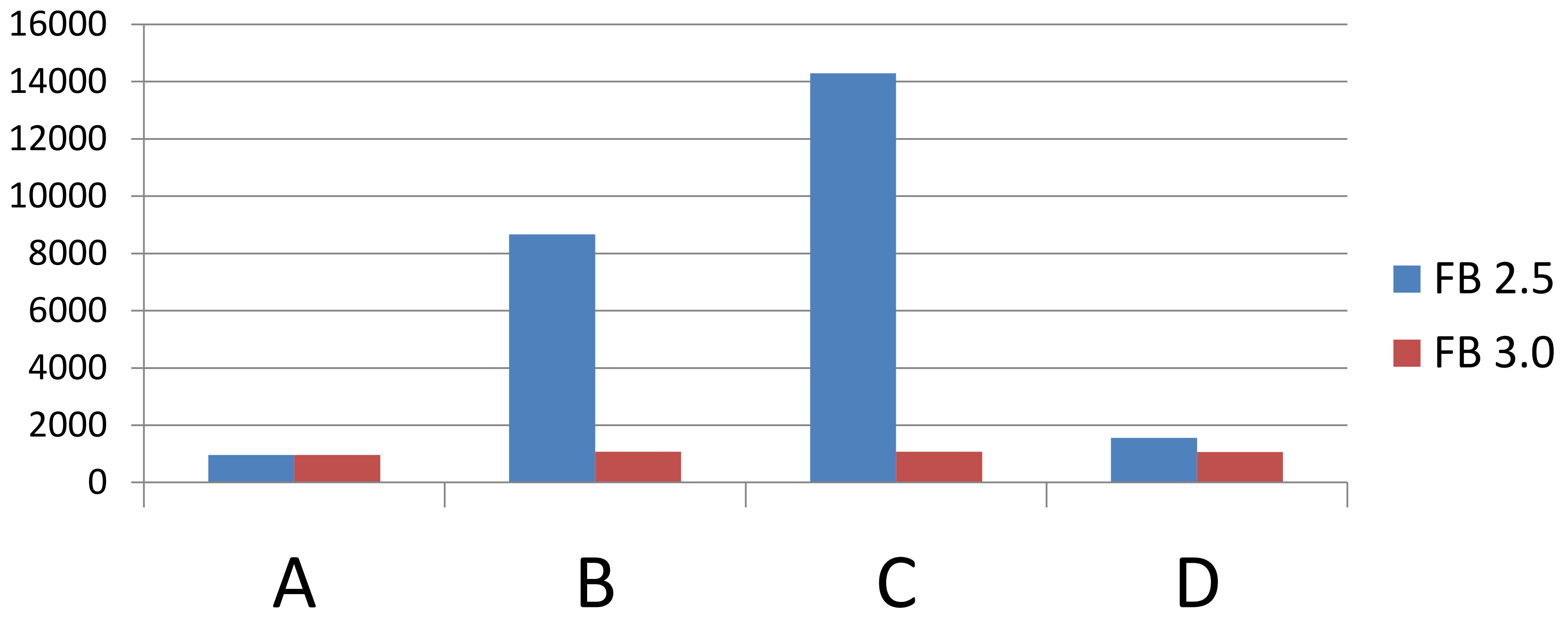
Memory buffers = 256

Reads from disk to cache = 1 066

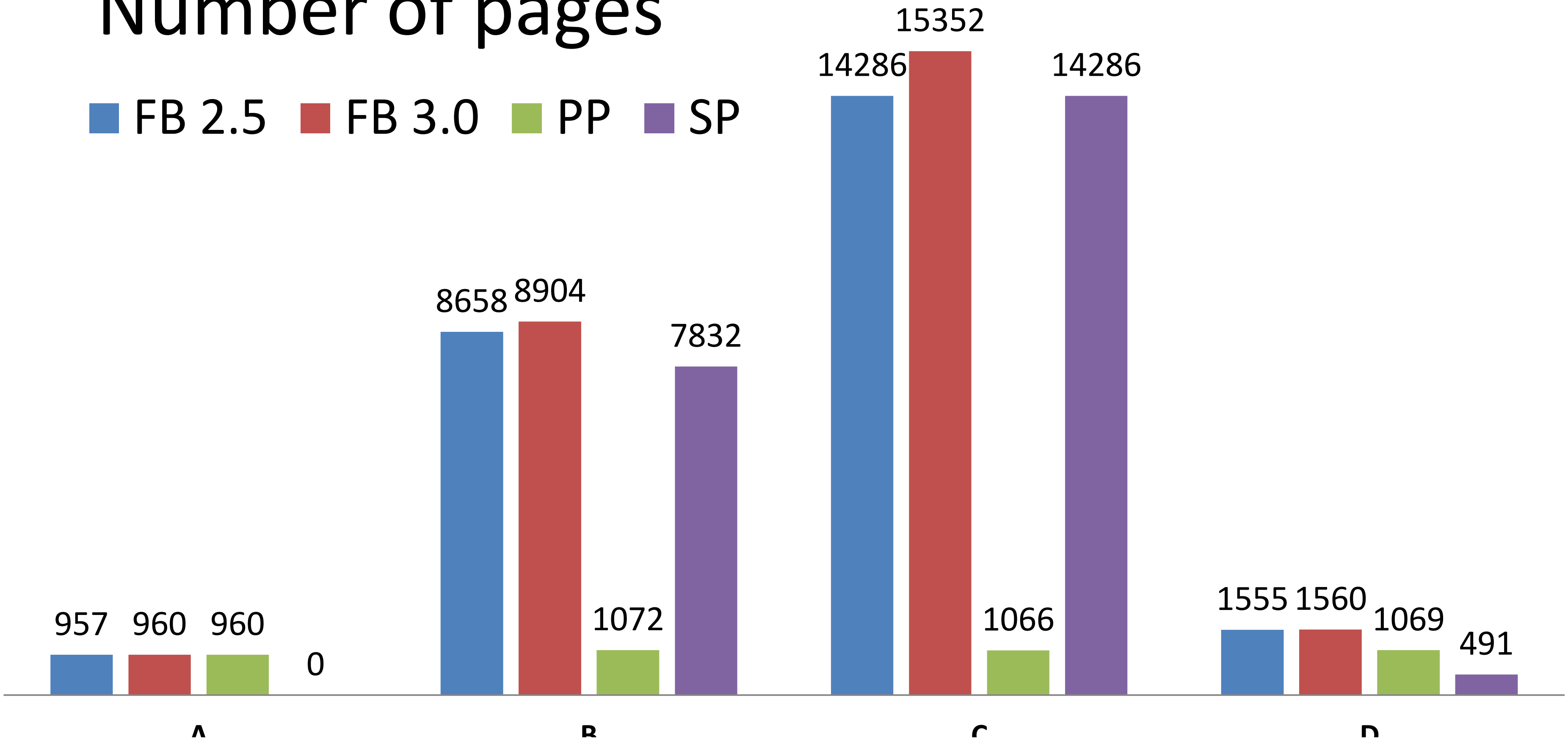
Writes from cache to disk = 0

Fetches from cache = 104 274

Page reads



Number of pages



Results

- Now blob record is placed at secondary page
- Blobs and versions may be mixed at secondary pages only
- Scanning data without blobs is faster and uses less I/O
- Level 0 in FB 3.0 by performance similar to blobs Level 1 in Firebird 2.5
- Scanning - 2x times less fetches
- Select count - 25% less fetches

Indices

- Index MAXSALX (2)
- **Root page: 324**, depth: 1, leaf buckets: 1, nodes: 31
- **Average node length: 14.74**, total dup: 5, max dup: 1
- **Average key length: 13.71**, **compression ratio: 1.37**
- **Average prefix length: 7.87**, average data length: 10.90
- **Clustering factor: 1, ratio: 0.03**
- Fill distribution:
- 0 - 19% = 1
- 20 - 39% = 0
- 40 - 59% = 0
- 60 - 79% = 0
- 80 - 99% = 0

Node, key, prefix

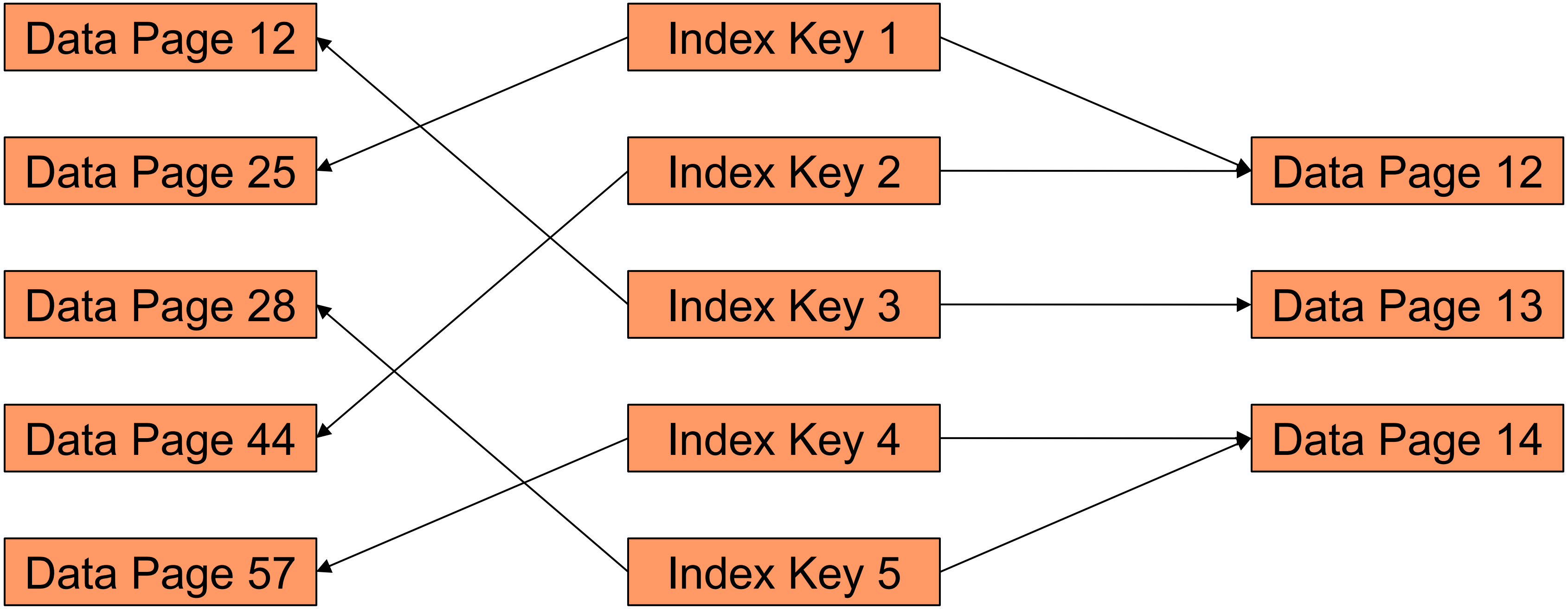
- **Average node length: 14.74**
- **Average key length: 13.71**, **compression ratio: 1.37**
- **Average prefix length: 7.87**, average data length: 10.90

- Node – prefix + key + record number
- Key – indexed data (column value)
- Compression –

board	0 5 board
boarding	5 3 ing
boarded	5 2 ed

- Sequential numbers may be compressed up to 8 times in comparison with GUIDs

Clustering Factor



Data Page 12

Data Page 25

Data Page 28

Data Page 44

Data Page 57

Index Key 1

Index Key 2

Index Key 3

Index Key 4

Index Key 5

Data Page 12

Data Page 13

Data Page 14

Bad Clustering Factor = 5
guid primary key

Good Clustering Factor = 3
int/bigint primary key

Example: Clustering factor: 1066, ratio: 0.01

- nodes: 100000
- Primary pages: 1066
- Ratio = Clustering factor / Nodes = 1066/100000 = 0.01
  - Ratio * keys in range = future DP reads
- The best Clustering factor – equal to the number of primary data pages

Example: Clustering factor: 1066, ratio: 0.01

- Clustering factor – jumps to different primary data pages while walking through index
  - 1066 with 1066 primary pages means the best
- Another index may have worse clustering factor
  - For example, 2132, i.e. will be 2x primary page reads
- Cache size – will these data pages fit, or not? (will be re-read from disk)

Clustering factor

- Clustering factor closer to the primary data pages number – good
- Ratio – less is better
- **How clustering factor affects performance?**
  See below

PLAN ELEMENTS

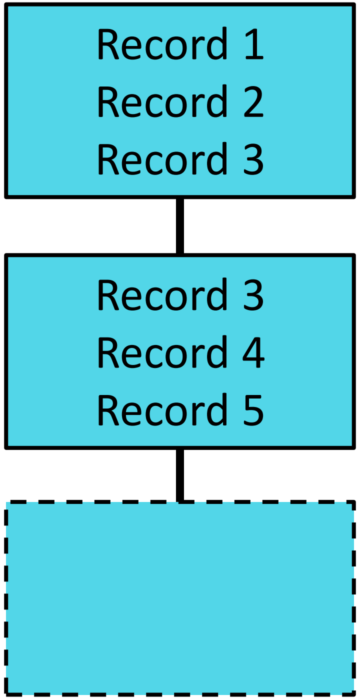
Plan elements

- tablename NATURAL
- tablename INDEX indexname
- Tablename ORDER indexname
- JOIN
- HASH JOIN
- SORT
- SORT MERGE

PLAN (TABLE NATURAL)

- **select * from employee**
  PLAN (EMPLOYEE NATURAL)

- The fastest way to read data

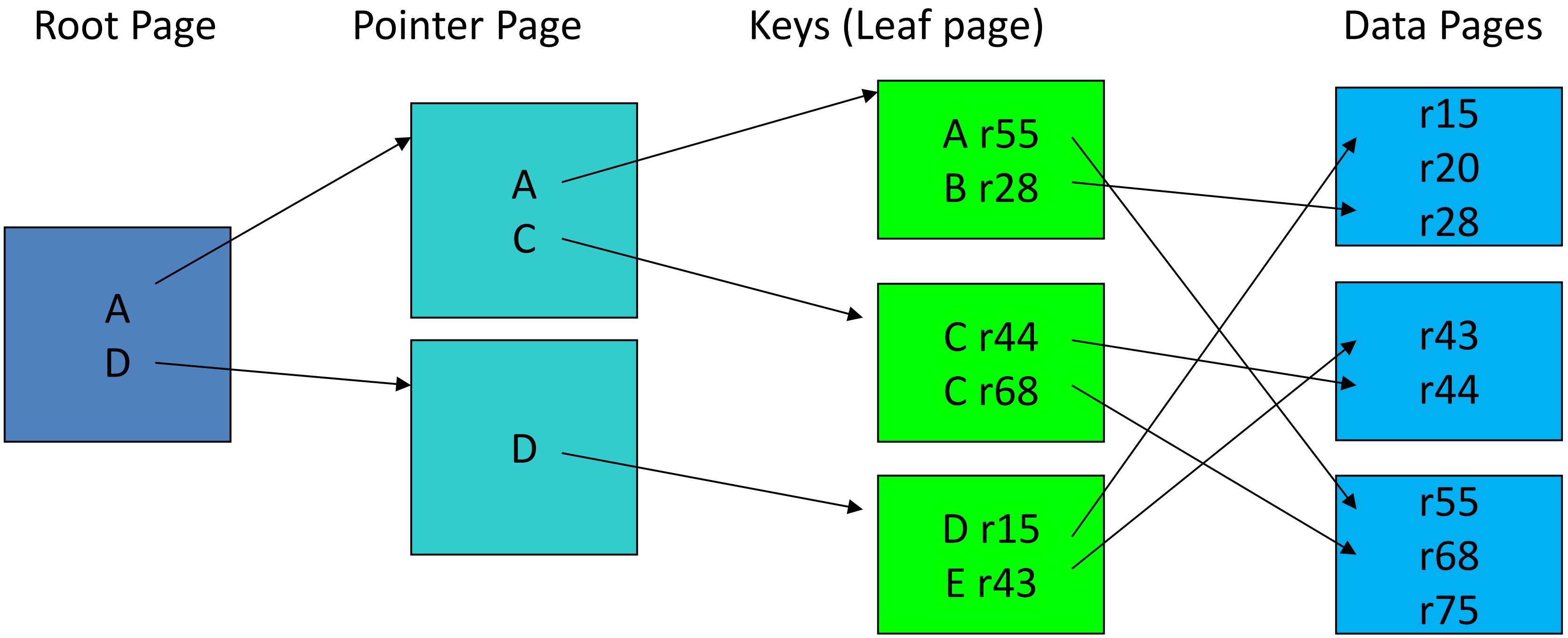


PLAN (TABLE INDEX *indexname*)

- Search for first key applying to condition
- Collect all row numbers for keys, that applying to condition
- *Sort array of row numbers*
- Fetch records from sorted array of row numbers

- **select * from employee**
  where emp_no > 5
  PLAN (EMPLOYEE INDEX (RDB$PRIMARY7))

Index -> Table

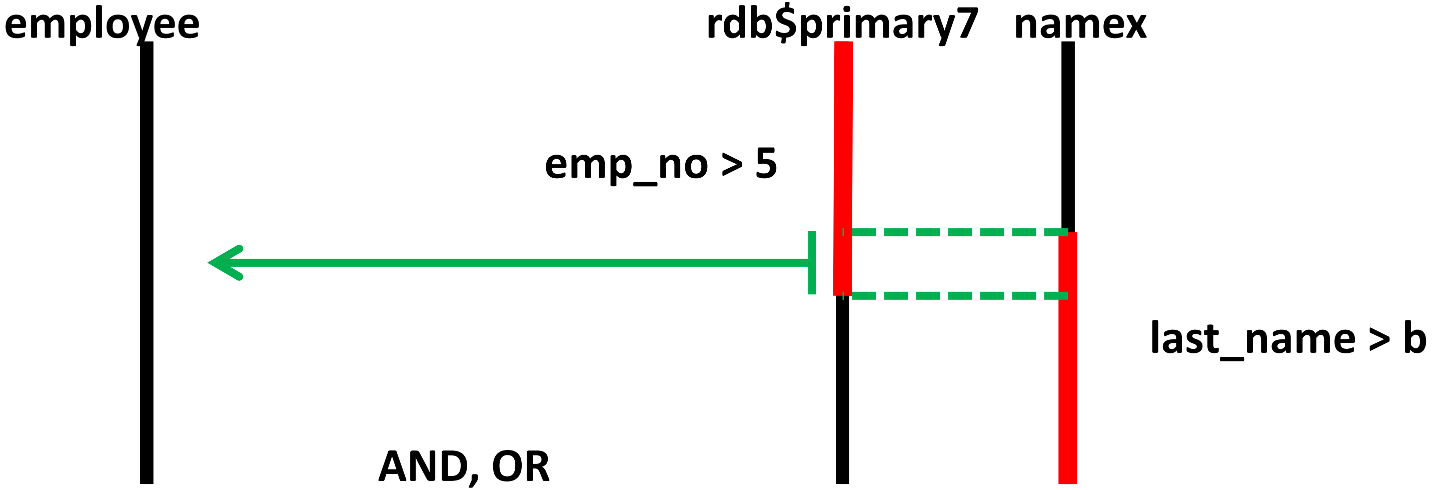

Root Page
Pointer Page
Keys (Leaf page)
Data Pages
A
D
A
C
D
A r55
B r28
C r44
C r68
D r15
E r43
r15
r20
r28
r43
r44
r55
r68
r75

How to force optimizer to use index

- We know that all employees have emp_no > 0. Then…
- **select * from employee**
  where emp_no > 0
  PLAN (EMPLOYEE INDEX (RDB$PRIMARY7))
- But,
  - All index pages will be scanned to get row numbers of all keys
  - data pages will be scanned too (to read records)
  - Result – bigger page I/O
- Sometimes this trick allows to change PLAN (and query speed)

Index bitmap merge

- **select * from employee**
  where emp_no > 5 and last_name > 'b'
  PLAN (EMPLOYEE INDEX (RDB$PRIMARY7, NAMEX))



- **select * from employee**
  where emp_no > 5 and last_name > 'b'
  PLAN (EMPLOYEE INDEX (RDB$PRIMARY7, NAMEX))

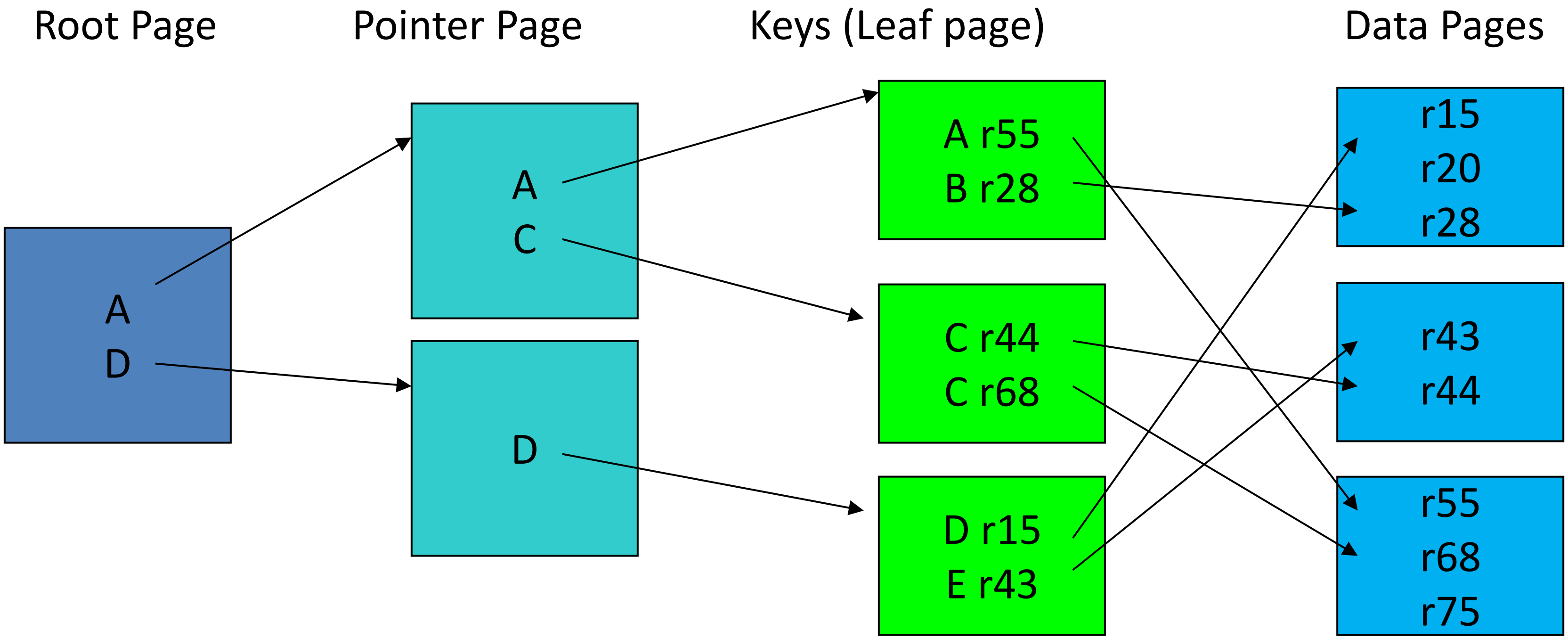
- **You will not get that plan in Firebird 3 in employee.fdb**
- Because optimizer eliminates indices on small tables
- Real plan:
  PLAN (EMPLOYEE INDEX (RDB$PRIMARY7))

PLAN (TABLE ORDER INDEX)

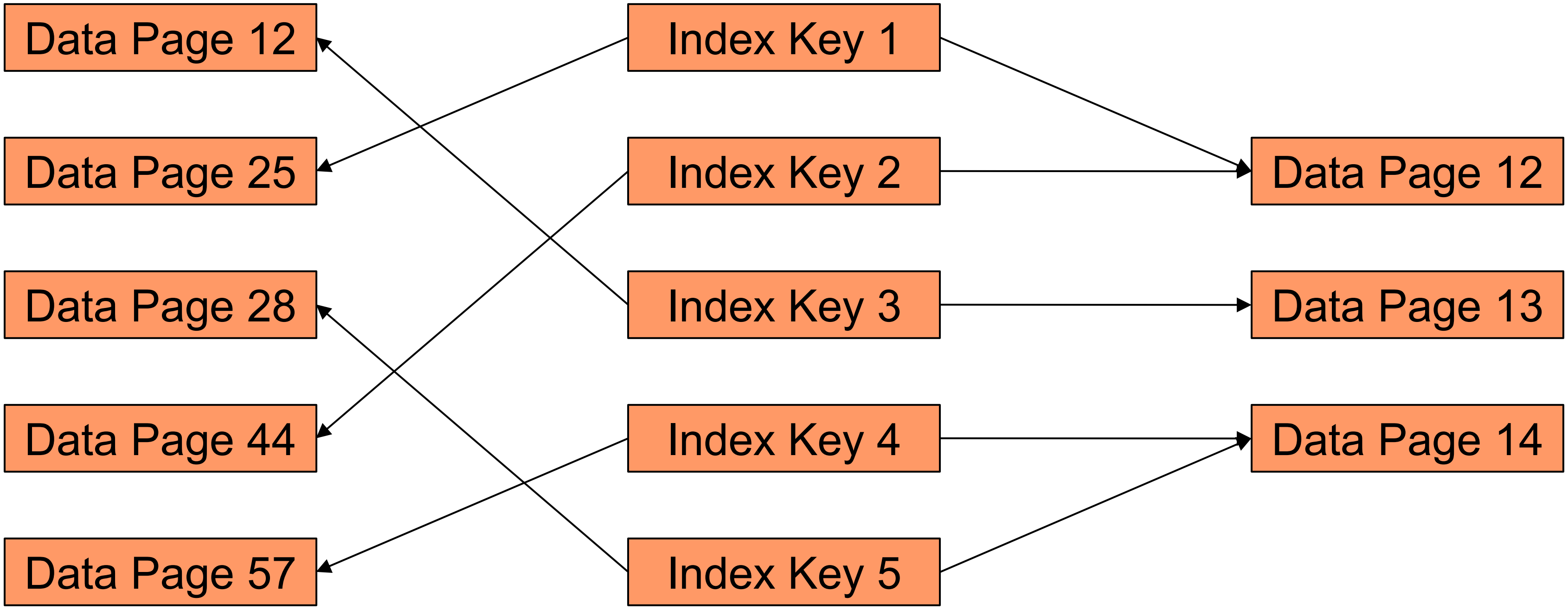
- Table walk by index order
- **select * from employee**
  order by last_name
  PLAN (EMPLOYEE ORDER NAMEX)

- Stays on first key (or key by where condition)
- Read record
  - apply filter, if any
- Goto next key
- Read record

Index -> Table


Root Page
Pointer Page
Keys (Leaf page)
Data Pages
A
D
A
C
D
A r55
B r28
C r44
C r68
D r15
E r43
r15
r20
r28
r43
r44
r55
r68
r75

Clustering Factor



Bad Clustering Factor
guid primary key

Good Clustering Factor
int/bigint primary key

Summary for *table ORDER index*

- Returns first row very quickly
- Jumping by data pages
  - Causing pages dropping from cache, if cache size can’t fit all data pages read
- Index Clustering factor
  - Order of keys corresponding to records
  - Firebird 3 – between pages and rows (less is better)
- Example

Index Order Example

- **select count(*) from table** (14mln records)
  Execute time = 42s 500ms
  Buffers = 2048
  Reads = 118 792
  Fetches = 28 814 893

Can be used to check disk performance
pages * page_size / sec = 43mb/sec

- **select a, count(a) from table**
  group by a
  PLAN (TABLE ORDER A)
  Execute time = 45m 55s 469ms
  Reads = 3 733 434
  each page was read from disk to cache 31 times
  Fetches = 42 869 143

- **select a from table**
  order by a
  PLAN (TABLE ORDER A)

  Execute time = 63ms
  Buffers = 2 048
  Reads = 48
  Fetches = 12 495

- if user will press Ctrl/End, it will take 3 mln reads and 45 minutes to get to the last row

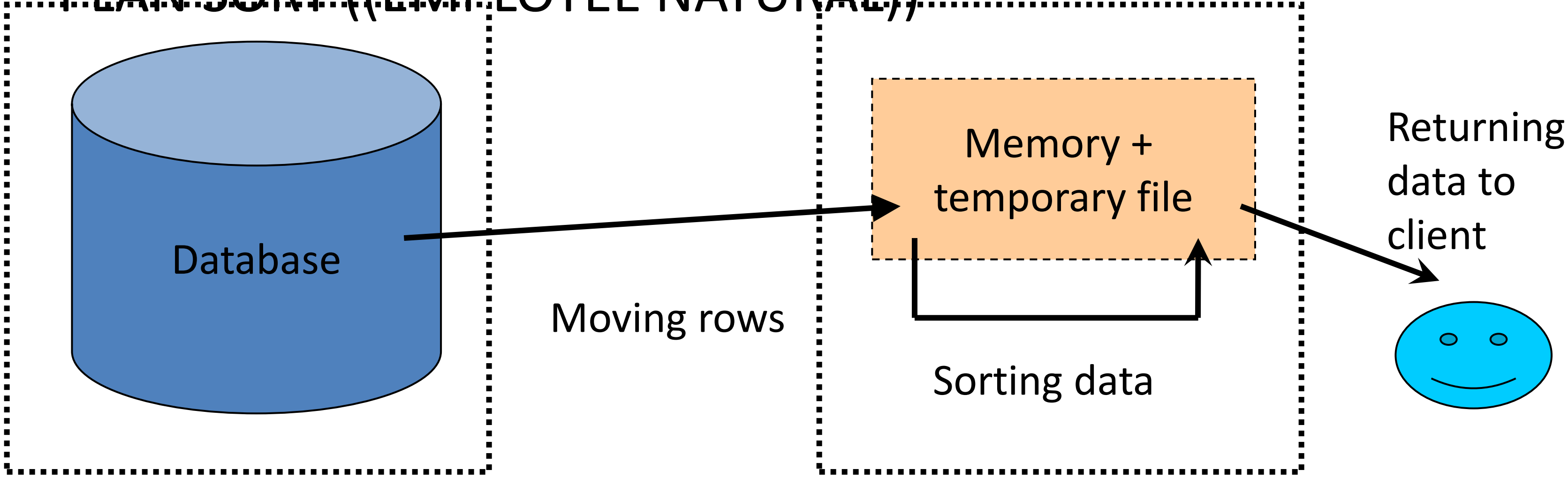
table ORDER index notes

- Affects ORDER BY and GROUP BY
  - Difference is only between number of rows returned to the client
  - ! Group by may use another access method instead SORT, so do not use GROUP BY for ordering
- Lot of rows causes huge page I/O
- Quickly return first rows, takes long time to get to the last row
- Only one index can be used – order of fields, number of fields and order direction must correspond to index

PLAN SORT

- **select * from employee**
- **order by first_name**

PLAN SORT ((EMPLOYEE NATURAL))



Sort tuning

- firebird.conf
  - TempBlockSize = 1048576
    - May increase to 2 or 3mln bytes, but not to 16mb
  - TempCacheLimit = 67108864
    - SuperServer and SuperClassic. Classic = 8mb.
  - TempDirectories = c:\temp;d:\temp…

  - Classic – RAM Disk, point TempDirectories to RAM disk first, to hdd next
  - SC, SS – tune Temp* parameters

ORDER vs SORT

PLAN (TABLE ORDER A)
Execute time = 45m 55s 469ms
Buffers = 2 048
Reads = 3 733 434
Fetches = 42 869 143

PLAN SORT ((A NATURAL))
Execute time = 2m 5s 485ms
Buffers = 2 048
Reads = 118 757
Fetches 28 813 410

- Reads equal to the table size (select count(*))
- Takes 2 minutes, then ready to return the whole result without delay
- Temp file is deleted when last row fetched
- N of temp files = N of queries with plan sort
  - Need to monitor number of temp files and their size

- Average record length: 118.86, total records: 14 287 964
- Data pages: 120 408, average fill: 99%
- Primary pages: 120 408, secondary pages: 0, swept pages: 0

- Index BY_CZ (5)
  - Clustering factor: 2 196 857, ratio: 0.15
- Index MINS_CLIENT (0)
  - Clustering factor: 3 651 564, ratio: 0.26
- Index MINS_DATE (3)
  - Clustering factor: 7 755 573, ratio: 0.54
- Index MINS_NUMA (1)
  - Clustering factor: 8 242 351, ratio: 0.58

EXPLAIN PLAN

Old and new plan output

- ISQL
- set planonly;
- Old plan example:
  PLAN SORT (RDB$RELATIONS INDEX (RDB$INDEX_0))
- set explain;
- …

Old and new plan output

```
SELECT * FROM RDB$RELATIONS
WHERE RDB$RELATION_NAME > :a
ORDER BY RDB$SYSTEM_FLAG
```

PLAN SORT (RDB$RELATIONS INDEX (RDB$INDEX_0))

```
Select Expression
    -> Sort (record length: 484, key length: 8)
        -> Filter
            -> Table "RDB$RELATIONS" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_0" Range Scan (lower bound: 1/1)
```

Old and new plan output

- SELECT * FROM RDB$RELATIONS
- WHERE RDB$RELATION_NAME > :a
- ORDER BY RDB$SYSTEM_FLAG

- PLAN SORT (RDB$RELATIONS INDEX (RDB$INDEX_0))

```
Select Expression
    -> Sort (record length: 484, key length: 8)
        -> Filter
            -> Table "RDB$RELATIONS" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_0" Range Scan (lower bound: 1/1)
```

Old and new plan output

- SELECT * FROM RDB$RELATIONS
- WHERE RDB$RELATION_NAME > :a
- **ORDER BY RDB$SYSTEM_FLAG**

- PLAN SORT (RDB$RELATIONS INDEX (RDB$INDEX_0))

```
Select Expression
    -> Sort (record length: 484, key length: 8)
        -> Filter
            -> Table "RDB$RELATIONS" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_0" Range Scan (lower bound: 1/1)
```

Index Scan

- Lower bound
- Upper bound
- Full scan
- Unique scan

Composite indices

CREATE INDEX BY_AB ON MYTABLE (A, B)

SELECT * FROM MYTABLE
WHERE A = 1 AND B > 5
PLAN (MYTABLE INDEX (BY_AB))

A	B
1	1
1	2
1	3
2	1
2	2
2	3
3	1

- Second column sorted by groups, depending on first column values
- **where A > 1 and B > 5** will not use 2nd column
- **where A = 1 and B …** will use first and second column
- **where A = 1 and B = 5 and C …**

Index "RDB$INDEX_0" Range Scan (lower bound: 1/1)

- For composite indices > 1.
- First – how many segments were used
- Second – how many segments index have
- 1/3 – only one segment is used
- 2/3 – first 2 segments are used
- 3/3 – all segments are used
- 1/3 – very ineffective, 2/3 medium effective
  - Consider using single-column indices instead

Procedure plan

- Now – natural, instead of all plans for all queries

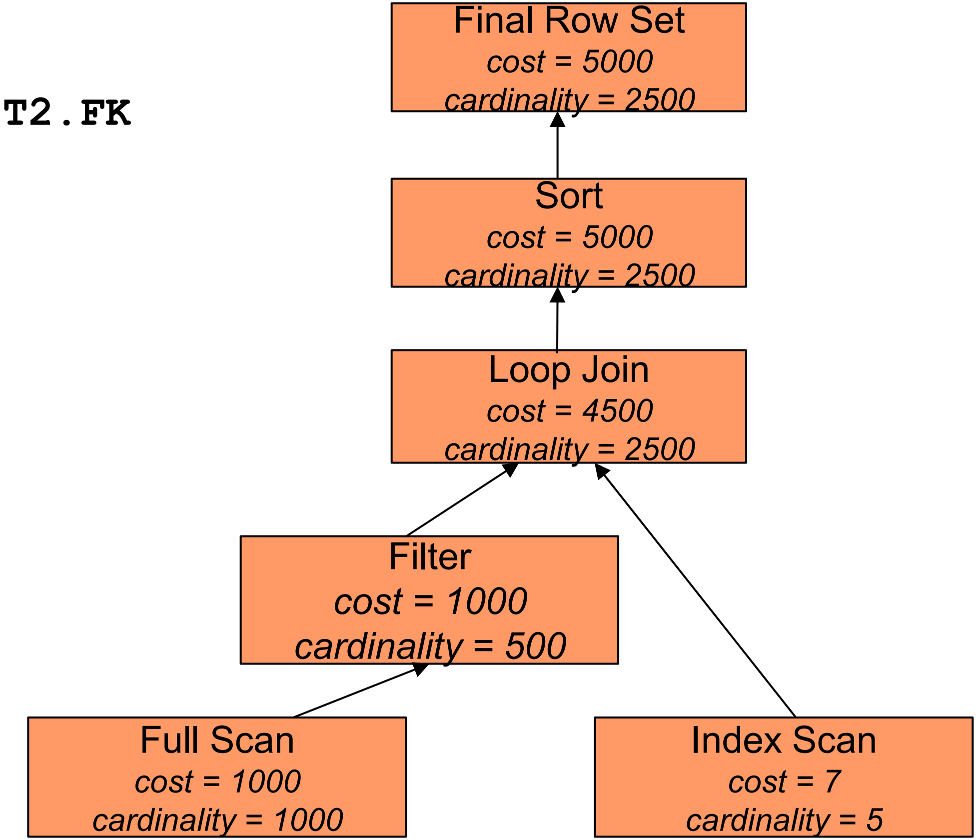
Cost estimation

- Cardinality – number of records in the table.
  - Computed by scanning pointer pages
- Selectivity – 1/(Keys – Total Dup)
  - The less is better. Number of unique key values = keys – total_dup

Cost estimation

```
SELECT *
FROM T1 JOIN T2 ON T1.PK = T2.FK
WHERE T1.VAL < 100
ORDER BY T1.RANK

PLAN
SORT (
  JOIN (
    T1 NATURAL,
    T2 INDEX (FK)
  ))
```



Final Row Set
cost = 5000
cardinality = 2500

Sort
cost = 5000
cardinality = 2500

Loop Join
cost = 4500
cardinality = 2500

Filter
cost = 1000
cardinality = 500

Full Scan
cost = 1000
cardinality = 1000

Index Scan
cost = 7
cardinality = 5

Table T1: base cardinality = 1000
Table T2: base cardinality = 5000
Index FK: selectivity = 0.001

EXPLAINED PLAN EXAMPLES

```
select * from rdb$relations
where rdb$relation_name > :a
PLAN (RDB$RELATIONS INDEX (RDB$INDEX_0))
```

```
Select Expression
    -> Filter
        -> Table "RDB$RELATIONS" Access By ID
            -> Bitmap
                -> Index "RDB$INDEX_0" Range Scan (lower bound: 1/1)
```

```sql
select * from a
where name > 'b' and a.id > 5
```

```
PLAN (A INDEX (ANAME, PK_A))

Select Expression
    -> Filter
        -> Table "A" Access By ID
            -> Bitmap And
                -> Bitmap
                    -> Index "ANAME" Range Scan (lower bound: 1/1)
                -> Bitmap
                    -> Index "PK_A" Range Scan (lower bound: 1/1)
```

select * from minutes

where code = '5' and zone > 5

PLAN (MINUTES INDEX (BY_CZ))

Select Expression
    -> Filter
        -> Table "MINUTES" Access By ID
            -> Bitmap
                -> Index "BY_CZ" Range Scan (lower bound: **2/2**, upper bound: **1/2**)

select * from rdb$relations
where rdb$relation_name > :a
order by rdb$relation_name

PLAN (RDB$RELATIONS ORDER RDB$INDEX_0)

Select Expression
    -> **Filter**
        -> Table "RDB$RELATIONS" Access By ID
            -> Index "RDB$INDEX_0" **Range Scan** (lower bound: 1/1)

! No “Bitmap” – index walk

```
select * from rdb$relations
where rdb$relation_name > :a
order by rdb$relation_name||’’
PLAN SORT (RDB$RELATIONS INDEX (RDB$INDEX_0))
```

```
Select Expression
    -> Sort (record length: 582, key length: 100)
        -> Filter
            -> Table "RDB$RELATIONS" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_0" Range Scan (lower bound: 1/1)
```

```
select e.last_name, p.proj_id
from employee e, employee_project p
where e.emp_no = p.emp_no
PLAN JOIN (P NATURAL, E INDEX (RDB$PRIMARY7))
```

```
Select Expression
    -> Nested Loop Join (inner)
        -> Table "EMPLOYEE_PROJECT" as "P" Full Scan
        -> Filter
            -> Table "EMPLOYEE" as "E" Access By ID
                -> Bitmap
                    -> Index "RDB$PRIMARY7" Unique Scan
```

select e.last_name, p.proj_id
from employee e left join employee_project p
on e.emp_no = p.emp_no
where p.emp_no is null
PLAN JOIN (E NATURAL, P INDEX (RDB$FOREIGN15))

```
Select Expression
    -> Filter
        -> Nested Loop Join (outer)
            -> Table "EMPLOYEE" as "E" Full Scan
            -> Filter
                -> Table "EMPLOYEE_PROJECT" as "P" Access By ID
                    -> Bitmap
                        -> Index "RDB$FOREIGN15" Range Scan (full match)
```

```
select e.* from employee e, employee_project p
where e.emp_no+0 = p.emp_no+0
PLAN HASH (E NATURAL, P NATURAL)
```

```
Select Expression
    -> Filter
        -> Hash Join (inner)
            -> Table "EMPLOYEE" as "E" Full Scan
            -> Record Buffer (record length: 25)
                -> Table "EMPLOYEE_PROJECT" as "P" Full Scan
```

select * from employee

where (emp_no = :param) or (:param is null)

where (emp_no = :param) or (:param = 0)

Old plan

PLAN (EMPLOYEE NATURAL)

New plan

PLAN (<u>EMPLOYEE NATURAL, EMPLOYEE INDEX (RDB$PRIMARY7)</u>)

Plan change at runtime

```
Select Expression
    -> Filter
        -> Condition
            -> Table "EMPLOYEE" Full Scan
            -> Table "EMPLOYEE" Access By ID
                -> Bitmap
                    -> Index "RDB$PRIMARY7" Unique Scan
```

```
select * from employee
where last_name = 'b'
order by first_name
PLAN (EMPLOYEE ORDER NAMEX)
```

```
Select Expression
    -> Filter
        -> Table "EMPLOYEE" Access By ID
            -> Index "NAMEX" Range Scan (partial match: 1/2)
```

select * from employee where emp_no in (1, 2, 3)

PLAN (EMPLOYEE INDEX (RDB$PRIMARY7, RDB$PRIMARY7, RDB$PRIMARY7))

```
Select Expression
    -> Filter
        -> Table "EMPLOYEE" Access By ID
            -> Bitmap Or
                -> Bitmap Or
                    -> Bitmap
                        -> Index "RDB$PRIMARY7" Unique Scan
                    -> Bitmap
                        -> Index "RDB$PRIMARY7" Unique Scan
                -> Bitmap
                    -> Index "RDB$PRIMARY7" Unique Scan
```

- Field in (1,2,3)
  - Uses index 3 times – one bitmap, 3 scans
- Field+0 in (1,2,3)
  - Turns index usage off, completely
- Field+0 in (1, 2,3) and (field between 1 and 3)
  - Turns index on back, range scan once, to avoid natural scan

ANOTHER FIREBIRD 3 AND 4 OPTIMIZER FEATURES

- Stream materialization (caching)
  - allows to avoid re-reading the same data from tables (for non-correlated streams)
  - currently used only for hash joins, to be used for subqueries too
- Hash join
  - join algorithm for non-indexed correlation
  - usually performs better than merge join
  - can be used instead of nested loops to avoid repeating reads of the same rows (in the future)

- FULL JOIN improvements
  - reimplemented as «semi-join union all anti-join»
  - can use available indices now
- Conditional streams
  - allow to choose between possible plans at runtime
  - currently used only for(FIELD = :PARAM OR :PARAM IS NULL)

- Improved ORDER plan implementation
  - avoid bad plans like «A ORDER I INDEX(I)», use simple «A ORDER I» with a range scan instead
  - allow ORDER plan for «WHERE A = 0 ORDER BY B» if compound index {A, B} exists

- Implicit FIRST ROWS / ALL ROWS hints
  - whether you need to fetch the first rows faster (e.g. interactive grids) or the complete result set –
    choose ORDER or SORT
  - currently used internally for queries with FIRST, EXISTS, ANY
  - prefers ORDER plan, affects join order
  - explicit FIRST/ALL ROWS hints for other query types will appear in v4

- Faster prepare for big tables
    - sampling PP instead of reading them all
    - being field tested
- Misc improvements
    - improve some cases of ORDER plan usage in complex queries
    - better plans for LEFT JOIN and UNION used together
    - optimize SORT plan for FIRST ROWS strategy

Planned for v 4

- HASH/MERGE for outer joins
- Execute EXISTS/IN as semi-join
- LATERAL joins
- More optimizer statistics and its background update

Thank you!

- Contacts:

www.ib-aid.com

support@ib-aid.com