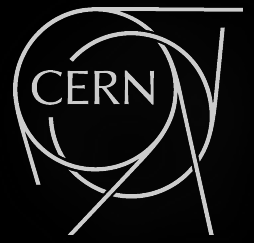


Top-level DB design for Big Data in ATLAS Experiment at CERN

Petya Vasileva, Gancho Dimitrov, Elizabeth Gallas

About Us

Elizabeth

A physicist by training and a database developer by experience: Working with DBAs to design and optimize schemas and applications to suit the needs of large scientific experiments.

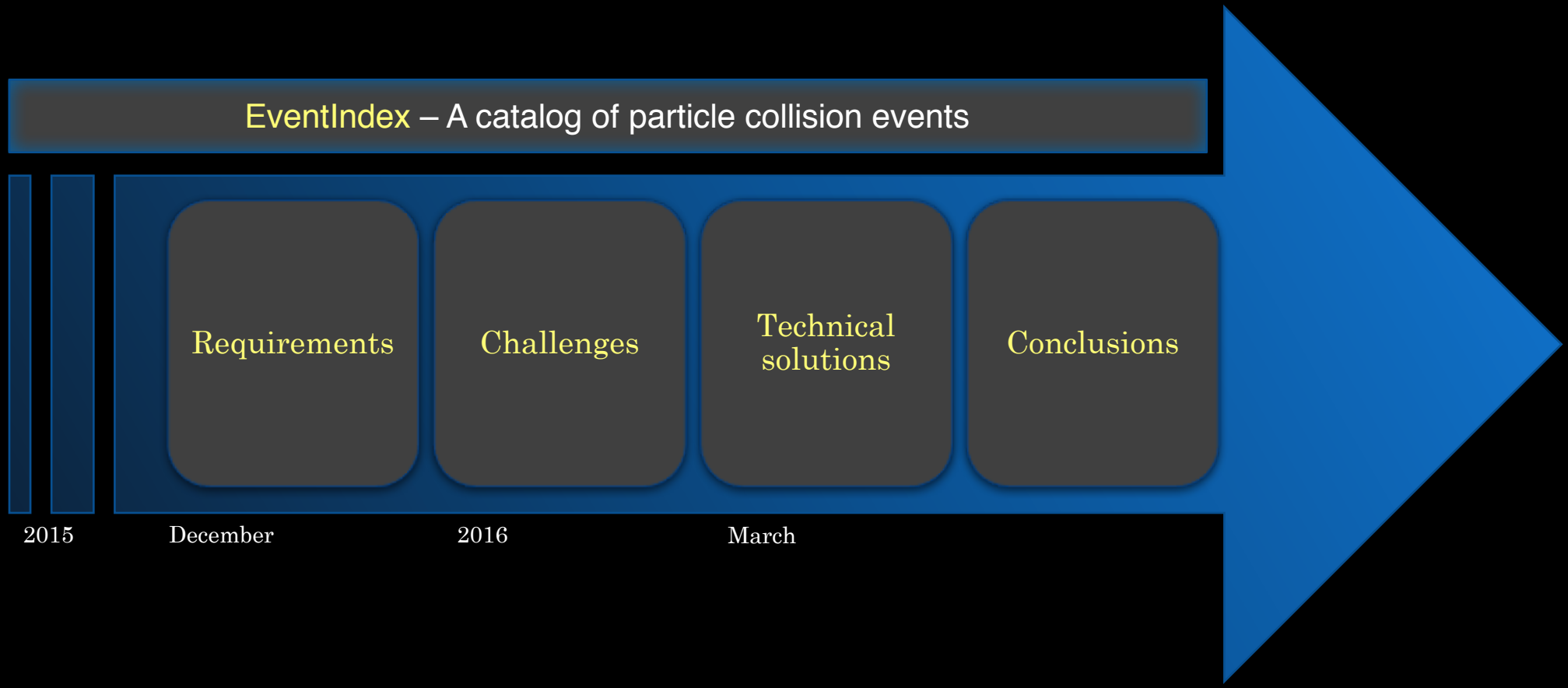
Petya

A software developer currently focusing on implementing applications which help DBAs and developers in their everyday work. She is also involved in providing support for the databases at ATLAS.

Gancho

Has a liaison role between the CERN/IT database services and support group and the ATLAS database developers community. Main focus is on database schemes design, data management and performance tuning.

Outline

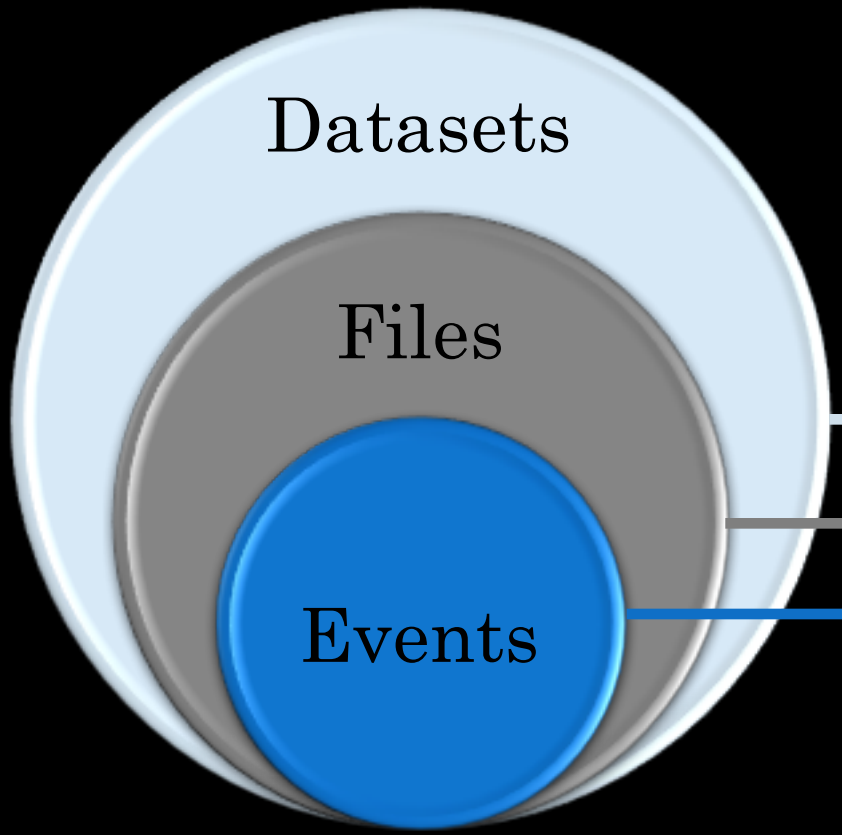

EventIndex – A catalog of particle collision events
Requirements
Challenges
Technical solutions
Conclusions
2015
December
2016
March

Oracle RDBMS and HW specs

Main database role	Post data-taking analysis
Oracle version	11.2.0.4 (DB in force logging mode)
# DB nodes	3
DB volume	37 TB
# DB schemes	172
HW specs	CPU Intel E5-2630 v4 @ 2.2GHz - 20 cores per node RAM 512 GB Per host 2 x 10 GigE for storage and cluster access NetApp NAS storage with 1.5 TB SSD cache

EventIndex
Basic information

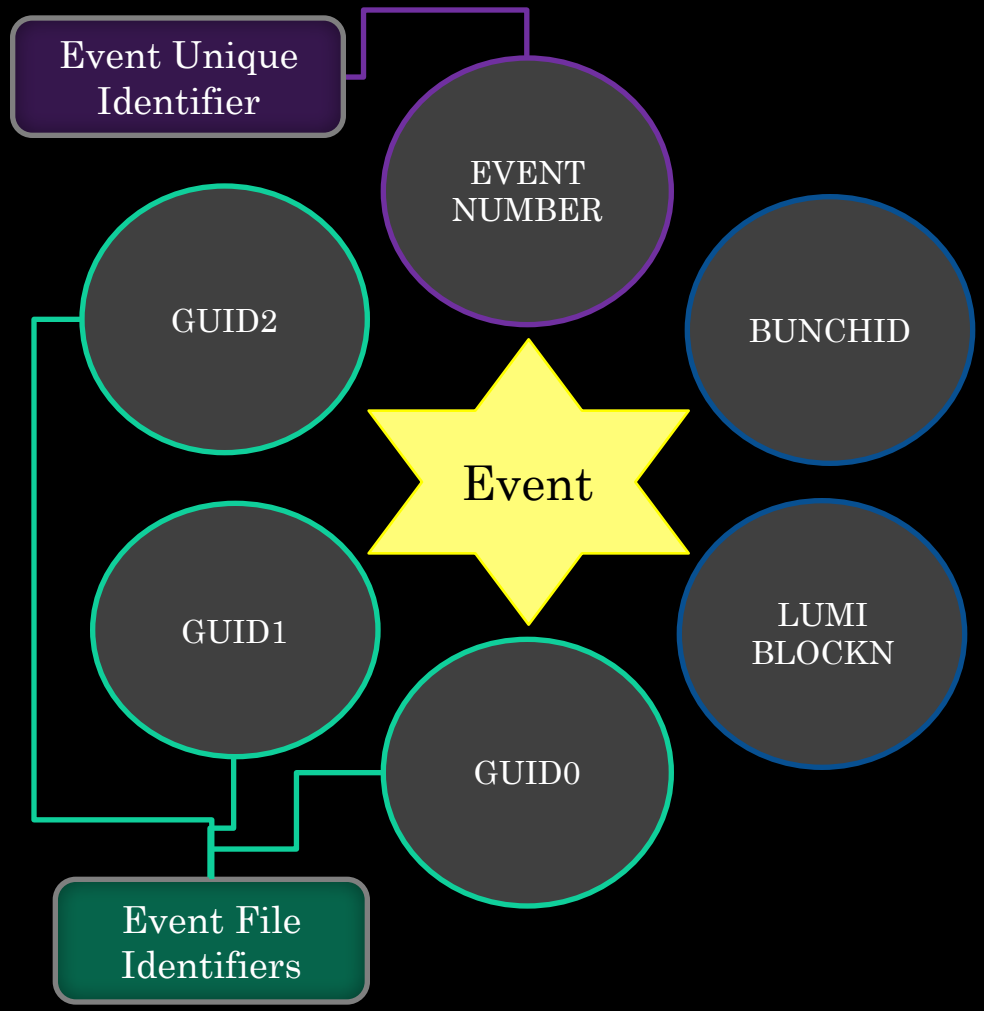
Particle collisions data



EventIndex is a catalogue containing information about the basic properties of event data as well as references to its storage location.

- Files are grouped into datasets
- Events are grouped into files
- Particle collisions are called **Events**

Event attributes



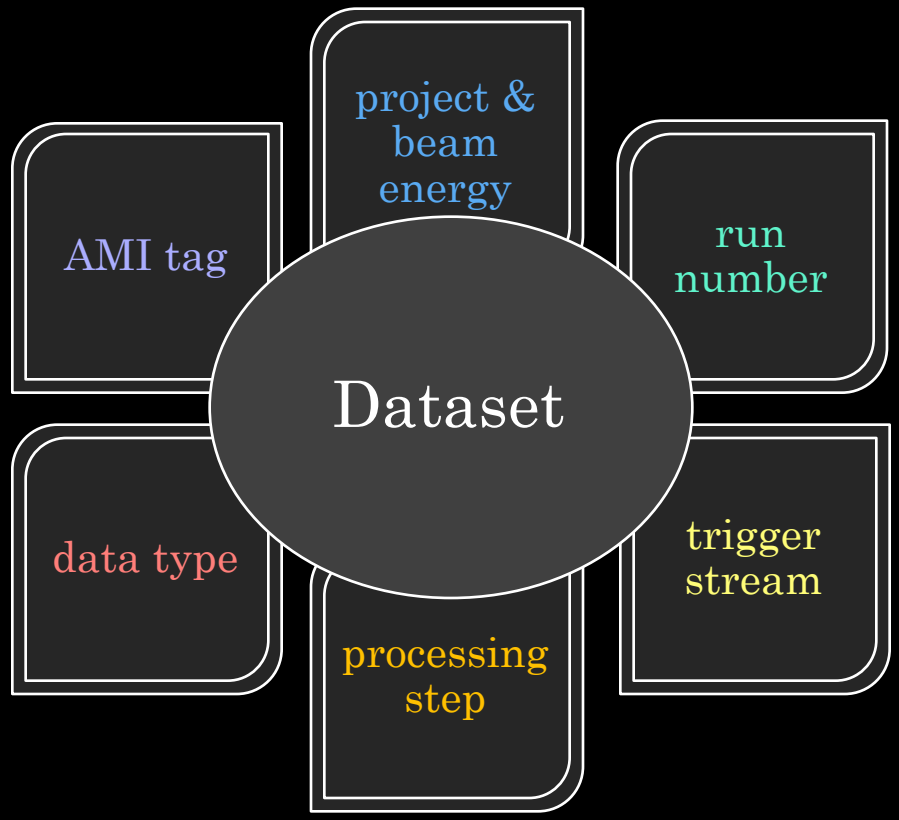
- Event numbers within a dataset are unique identifiers of particles collision event
- File identifier : GUID (***G**lobally **U**nique **Id**entifier)*
  - GUIDs are usually stored as 128-bit values, and are commonly displayed as 32 hexadecimal digits with groups separated by hyphens, such as:

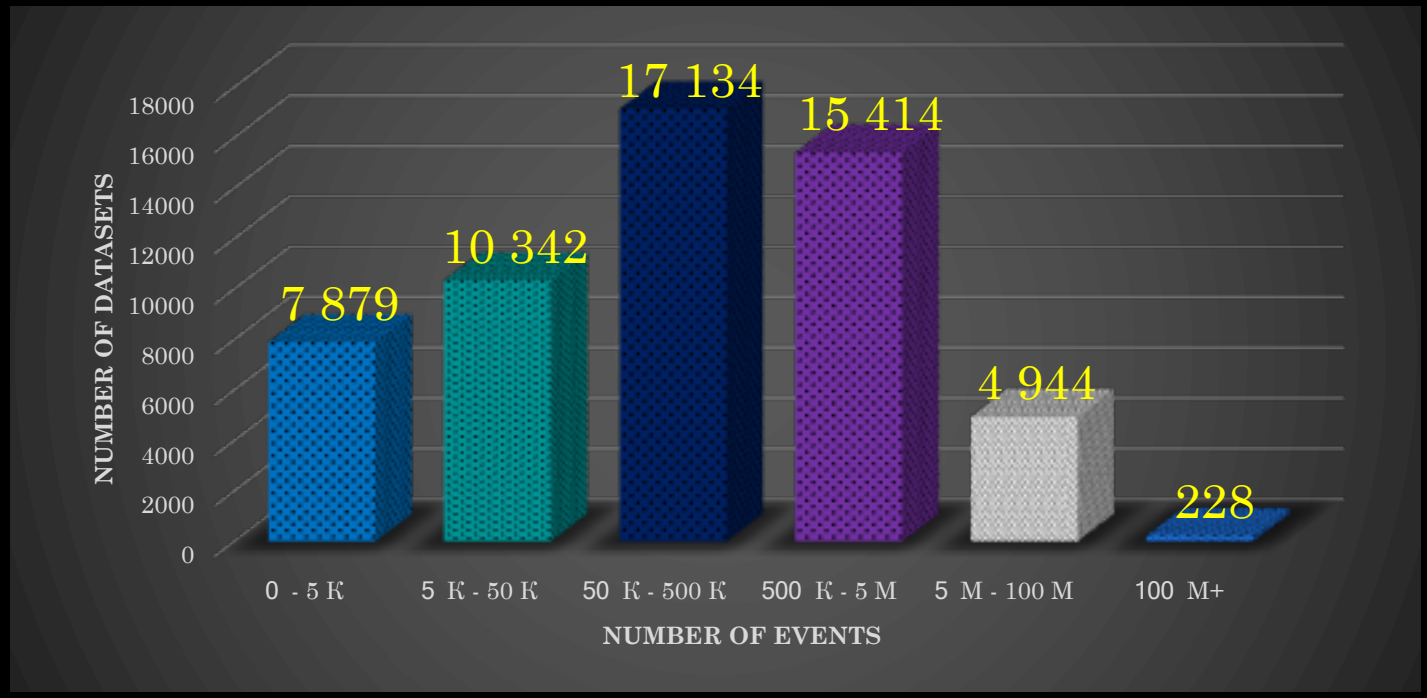
GUID example:

21EC2020-3AEA-4069-A2DD-08002B30309D

Dataset attributes

A unique dataset name is composed by 6 attributes
Example: data17_900GeV.00340308.physics_CosmicCalo.merge.AOD.f894_m1902

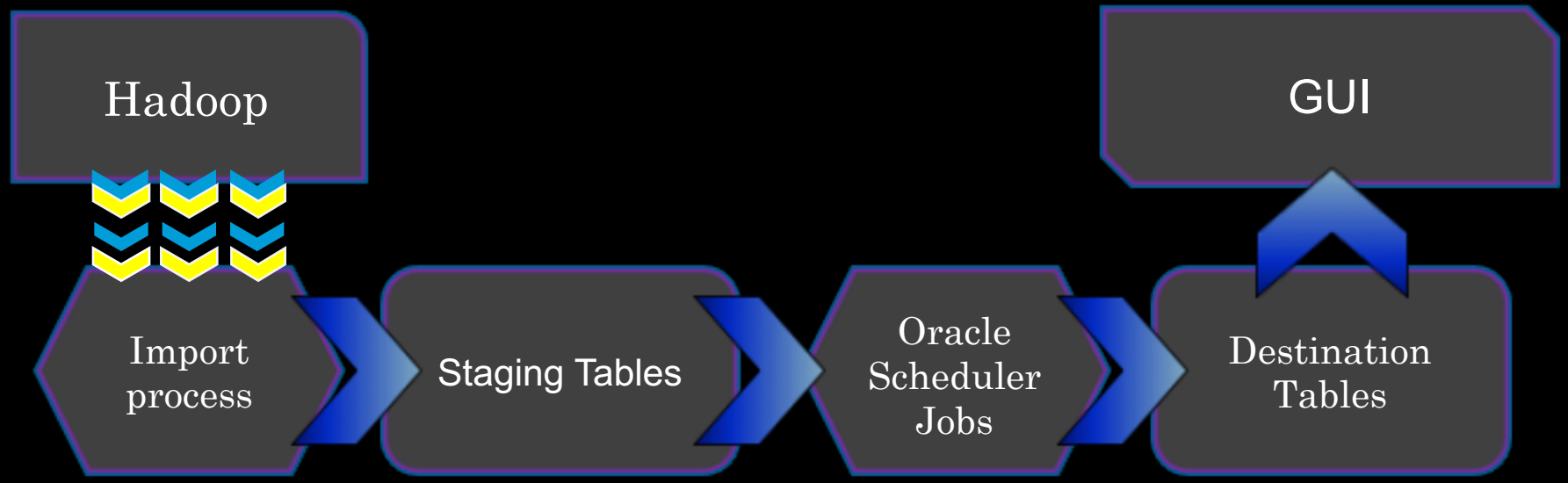




Expected datasets per year > 25 000

Dataflow

- Data is loaded from a Hadoop system
- All data is dumped into a plain table – no constraints, no indices
- Oracle jobs verify, optimize & copy data to the destination tables



Project requirements

- Store large amount of rows – tens of billions per year
- Remove large amount of rows if dataset is obsolete or its content has to be replaced.
- Guarantee uniqueness of events within each dataset
- Ensure fast and consistent data upload of large amount of rows (10s-100s million of rows)
- Crosscheck data with other systems
- Detect rows (events) duplications
- Retrieve file identifiers (GUIDs) in fraction of a second

Challenges

1. Large amount of rows require large disk volume plus significant index overhead
2. Achieve high-speed data load & reduce undo and redo generation
3. Automatic handling in case of uniqueness violation
4. Provide simple way for querying events data
5. Remove large amount of rows with minimum DB footprint
6. Have adequate table stats gathering policy
7. Optimize data retrieval & guarantee queries execution plan stability

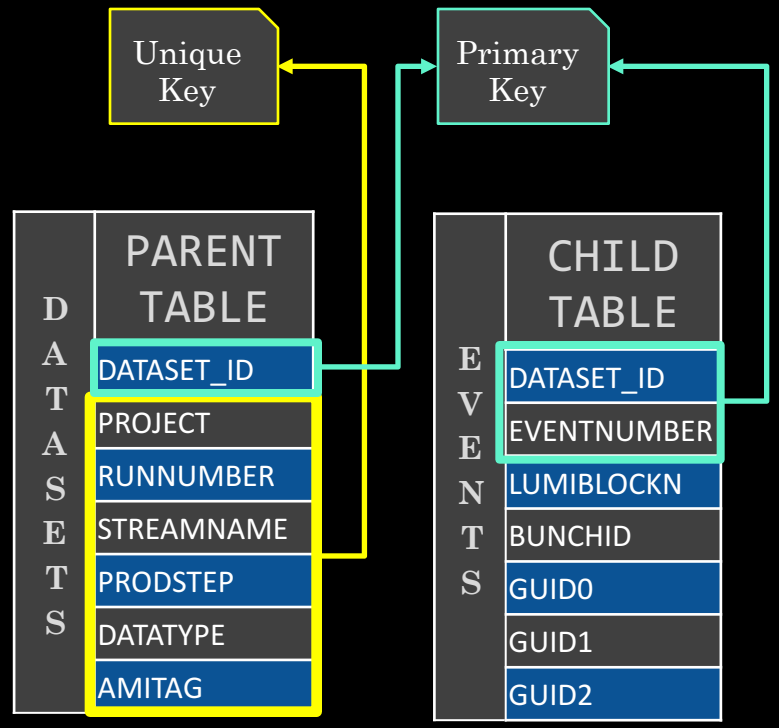
Challenge “1”
Significant amount of disk space

- On average 210 bytes per row
- 1B rows = 200 GB without the index overhead
- 25B rows/year = 5 TB/year without the index overhead

Used approach for challenge “1” (1)

Idea: study, test and get to acceptable level of data normalization to minimize data redundancy.

- “Parent table” with dataset definitions and “child table” having dataset content
- Dataset name uniqueness guaranteed by an Oracle unique constraint
- Dataset definition gets unique ID (dataset_id) from a DB sequence object



Improvement: the AVG row length is reduced from 210 bytes to 130 bytes.

- 80 bytes in the “parent table”
- 130 bytes in the “child table” (vast dataset range: from tens rows to 100s million rows per dataset)

Used approach for challenge “1” (2)

Idea: explore Oracle data types

- The three file identifiers (GUIDs) of each event is with fixed length : 36 chars.
- 3 GUIDs in row = 108 bytes when using Oracle “char” data type

GUID example: 21EC2020-3AEA-4069-A2DD-08002B30309D

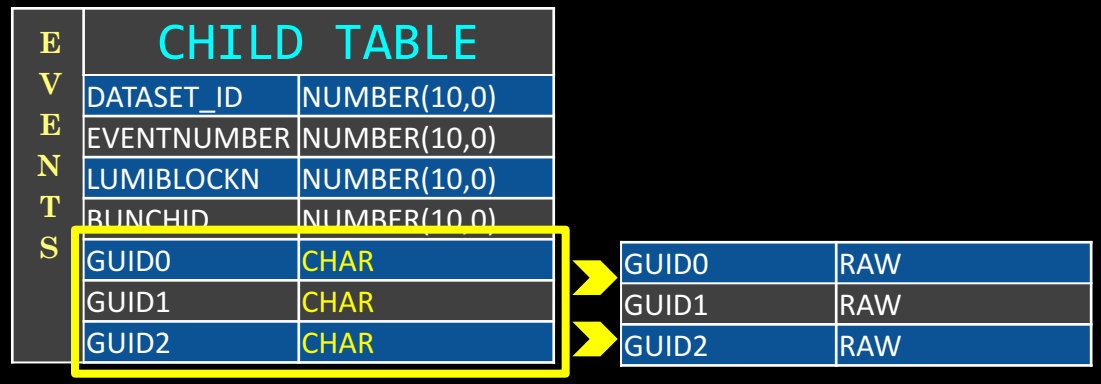
Question: Could we “shorten” the 130 bytes row by using other Oracle data types?

Answer: Yes, by taking advantage of the RAW data type

GUIDs type RAW 16 bytes < GUIDs type CHAR 36 bytes

Store GUIDs of data type RAW

```
HEXTORAW(REPLACE(GUID, '-', ''))
```



Improvement: AVG length of “child table” row is reduced to 70 bytes

Used approach for challenge “1” (3)

- **Idea:** explore the Oracle Basic and OLTP compression (de-duplication within data block of 8KB)

Test with 11 billion rows PCTFREE 0 setting:

Compression	AVG space per row	Table size
OLTP	19.6 bytes	201 GB
Basic	19 bytes	195 GB

Improvement : AVG length of “child table” row is reduced to 20 bytes.

Goal achieved: Disk space reduced significantly

Instead of 200 GB per 1B rows, only 20 GB are needed.

Challenge “2”
Transactions could be large.
How to achieve satisfactory speed?

- The logical unit of data is a “dataset” being a group of unique identifiers of particles collision event

- Dataset can have from few 10s of events to 100s million events (rows)

Approach for challenge "2"

Idea:

1. Insert raw events data into a staging table without any normalization, use "char" data type for the GUIDs (AVG row length = 210 bytes), no index overhead. Use list partitioning for having a dedicated partition per Dataset.
2. Use PLSQL procedure for loading data from the staging table to the "child table".

Test with a large transactions (100s millions rows)

Goal: Achieve consistency, high speed in rows insert, minimum used disk space, minimum undo and redo footprint on the DB.

Question: OLTP or Basic compression to be used?

EVENTS

CHILD TABLE	
DATASET_ID	NUMBER(10,0)
EVENTNUMBER	NUMBER(10,0)
LUMIBLOCKN	NUMBER(10,0)
BUNCHID	NUMBER(10,0)
GUID0	RAW
GUID1	RAW
GUID2	RAW

NO INDEX
NO CONSTRAINTS
LIST PARTITIONED BY DATASET_ID

EVENTS

STAGING TABLE	
DATASET_ID	NUMBER(10,0)
EVENTNUMBER	NUMBER(10,0)
LUMIBLOCKN	NUMBER(10,0)
BUNCHID	NUMBER(10,0)
GUID0	CHAR(36 BYTE)
GUID1	CHAR(36 BYTE)
GUID2	CHAR(36 BYTE)

Challenge "2": setup without compression

Test: Bulk insert into the "child table", no constraints and no compression

Test `INSERT /*+ append*/ INTO` child_table **not having PK**, **no FK,** no compression

Results:

1. Insert speed 202K rows/sec
2. Undo is not used

Test `INSERT /*+ append*/ INTO` child_table **having PK**, **no FK,** no compression

Results:

1. Insert speed 113K rows/sec
2. 1 GB undo used per 54 million rows (used only for the PK index build)

Challenge "2": OLTP compression (1)

Test "Child table" with OLTP compression.

Conventional insert

```
INSERT INTO child_table
SELECT … ,
     HEXTORAW(REPLACE(GUID0, '-', '')),
     HEXTORAW(REPLACE(GUID1, '-', '')),
     HEXTORAW(REPLACE(GUID2, '-', ''))
FROM staging_table
WHERE DATASET_ID = …;
```

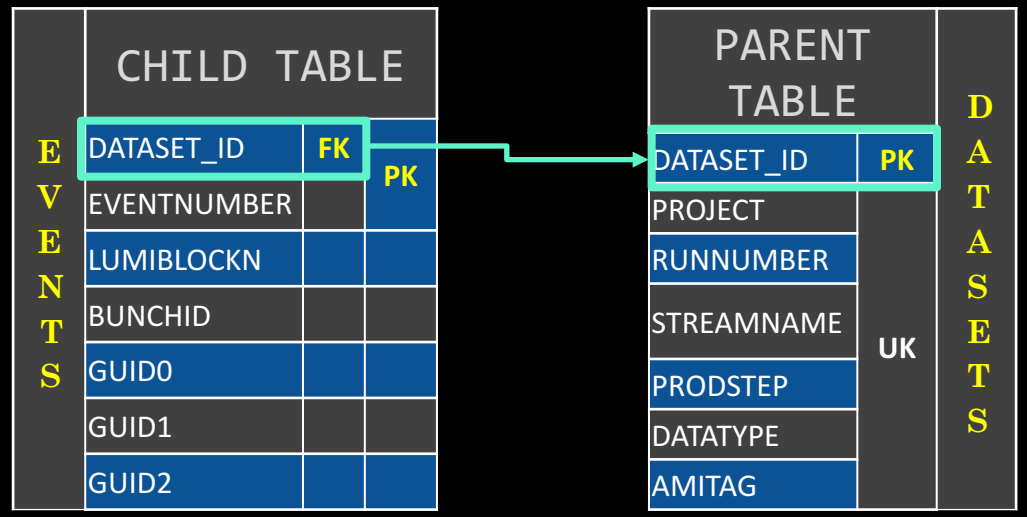
or

Bulk insert

```
INSERT /*+ append*/ INTO child_table
SELECT … ,
     HEXTORAW(REPLACE(GUID0, '-', '')),
     HEXTORAW(REPLACE(GUID1, '-', '')),
     HEXTORAW(REPLACE(GUID2, '-', ''))
FROM staging_table
WHERE DATASET_ID = …;
```

Same result:

1. Insert speed 7K rows/sec
2. 1GB undo per 3 million inserted rows
3. USED_UREC = 2x number of inserted rows because of the PK



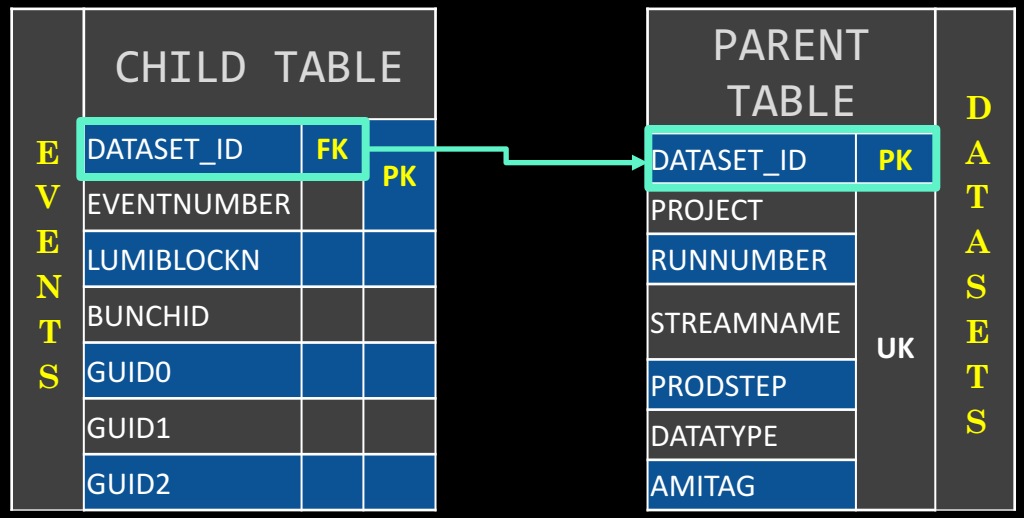
Challenge "2": Basic compression

Test "Child table" with Basic compression.

Test bulk insert (`INSERT /*+ append*/ INTO ...`)

Results:

1. Insert speed 13K rows/sec
2. 1GB undo per 6 million inserted rows
3. USED_UREC = 2x number of inserted rows because of the PK
4. Compression **does not kick in** (the AVG row length is 70 bytes). Why?



Note: for the largest dataset we have of 630 million rows, we would need 105 GB undo and about 14 hours to complete.

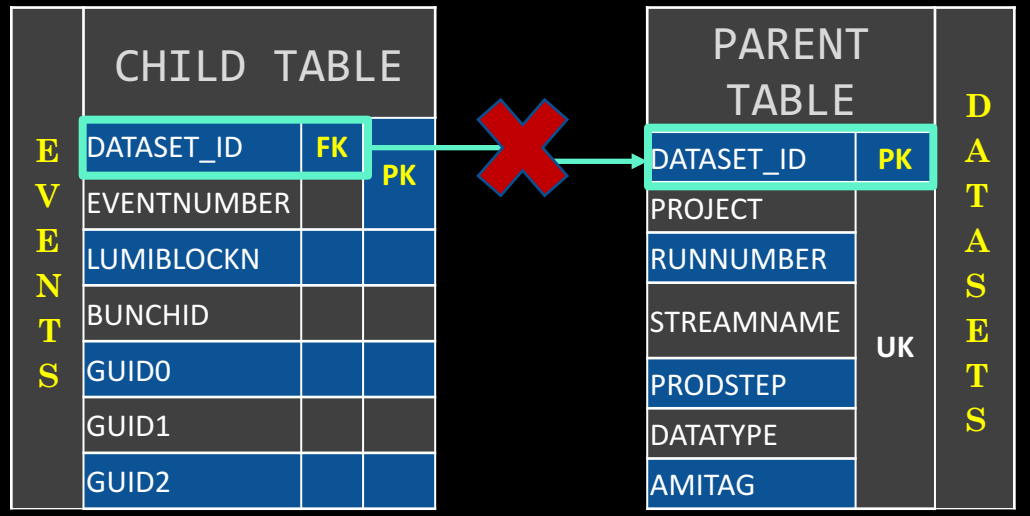
Challenge "2": Basic compression (2)

Question: Why the data compression does not kick in? Why still so much undo is used?

Test `INSERT /*+ append*/ INTO child_table` **having PK, no FK and with Basic compression**

Results:

1. Insert speed 130K rows/sec
2. 1 GB undo used per 54 million rows (used only for the PK index build)
3. Compression kicks in and the average space used per row is 20 bytes



Finding:

When having bulk insert on table with FK and Basic compression, Oracle silently does conventional insertion, does not compress the data and generates a lot of undo.

Challenge “2”: applied solution

✓ Goal achieved: data consistency, high speed in rows insert, minimum used disk space, minimum undo and redo footprint on the DB

Example: to load 100 million rows within a single transaction:

- Elapsed time is about 13-14 min (insert rate 120-130K rows/sec)
- Table segment about 2GB , index segment about 2GB
- Used undo < 2 GB

Compromises:

- **No parent-child table relationship enforcement via DB foreign key.**

This is acceptable as the data loading is done only by a PLSQL procedure that has the necessary pre-checks in. Other idea is to try with FK of type “deferred rely”.

- **Data load serialization because of the** `INSERT /*+ append*/ INTO child_table`

Only a single session can load data at a given time. This is acceptable because of the achieved high insert rate. Or potentially we could use parallelism in the bulk insert.

Challenge “3”
Automatic handling in case of keys uniqueness violation

- Application logic implies that raised “ORA-00001: unique constraint violated” error on the “child table” must not rollback the transaction, but any rows with duplicated key(s) have to be stored aside into a separate table.

Tested approach for challenge “3”

Idea: store aside the duplicated rows using the `LOG ERRORS INTO` … clause

Actions: Create errorlog table using the DBMS_ERRLOG.CREATE_ERROR_LOG method

```
INSERT /*+ append*/ INTO child_table

SELECT … FROM staging_table

LOG ERRORS INTO log_table REJECT LIMIT UNLIMITED;
```

Findings: that approach works as a standalone statement only in the SQL scope.

The " `LOG ERRORS INTO` … `REJECT LIMIT UNLIMITED` " does not work within a PLSQL proc.
The “ ORA-00001: unique constraint violated “ on the “Child table” PK is returned.

Tested approach for challenge “3” (2)

Idea: use the "ignore_row_on_dupkey_index" hint

```
ignore_row_on_dupkey_index(tab_name,table_PK_name)
                or
ignore_row_on_dupkey_index(tab_name(col1, col2))
```

```
INSERT /*+ ignore_row_on_dupkey_index(tab_name(col1, col2)) */
INTO child_table
SELECT … FROM staging_table
```

Findings:

- That approach works only for datasets with few hundred rows.
- For datasets with thousands of rows, Oracle corrupts the PK index structure.
- Counting rows in a dataset, computes more rows than inserted (details on the next slide).

Surprising result with ignore_row_on_dupkey_index…

Test: Find the number of occurrences of the 'unique' values

```
SELECT RUNNUMBER, EVENTNUMBER, count(*)
FROM child_table
WHERE DATASET_ID = …
GROUP BY RUNNUMBER, EVENTNUMBER
HAVING count(*) > 1;
Results:
    18395   2295389 2
    18395   2296722 3
    18395   2296824 3
    18395   2252272 4
    18395   2284925 2
```

Problem: deletion of rows from the dataset raised an error of corruption in the index

```
DELETE from child_table WHERE dataset_id = …

ORA-08102: index key not found, obj# 20667461, file 658, block 77946491 (3)
```

Resolution of challenge “3” (3)

Solution: sequence of actions encoded into a PLSQL procedure

Actions:

1. Check whether the dataset in the staging table has duplicates based on the destination table PK columns
2. Insert into the “child table” all rows that do not have duplication
3. Insert into a dedicated table all duplicated rows from the staging table
4. Insert into the “child table” a representative row from each group of duplication

Goal achieved: Duplicated rows are automatically detected and stored separately

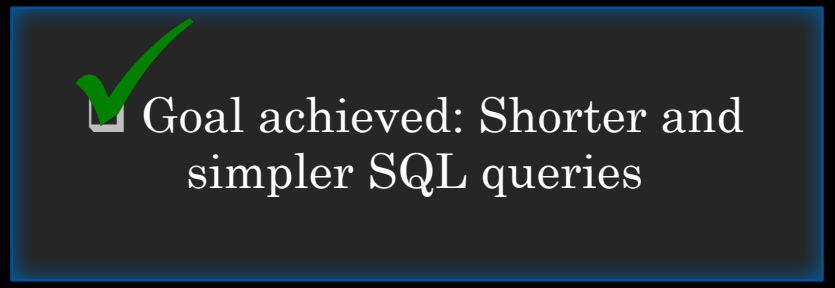
Challenge “4”
Provide simplification for the GUIDs retrieval

- GUID data types are of type RAW in order to safe space, however the application must get the GUID values as string

Approach for challenge “4”

Idea: Define virtual columns for the GUIDs on table level ready to be used in any query.
Advantages: values are computed on a fly, simplifies application queries (clearer SQL code)

```
EXAMPLE:
SELECT GUID0_CHAR, GUID1_CHAR, GUID0_CHAR
FROM child_table
```




```
CREATE TABLE child_table (
…
, GUID0              RAW(16)
 ,GUID1              RAW(16)
 ,GUID2              RAW(16)
 ,GUID0_CHAR as
(SUBSTR(RAWTOHEX(GUID0),1,8)||'-
'||SUBSTR(RAWTOHEX (GUID0),9,4)||'-
'||SUBSTR(RAWTOHEX (GUID0),13,4)||'-
'||SUBSTR(RAWTOHEX (GUID0),17,4)||'-
'||SUBSTR(RAWTOHEX (GUID0),21,12))
 ,GUID1_CHAR as
(SUBSTR(RAWTOHEX(GUID1),1,8)||'-
'||SUBSTR(RAWTOHEX (GUID1),9,4)||'-
'||SUBSTR(RAWTOHEX (GUID1),13,4)||'-
'||SUBSTR(RAWTOHEX (GUID1),17,4)||'-
'||SUBSTR(RAWTOHEX (GUID1),21,12))
 ,GUID2_CHAR as
(SUBSTR(RAWTOHEX(GUID2),1,8)||'-
'||SUBSTR(RAWTOHEX (GUID2),9,4)||'-
'||SUBSTR(RAWTOHEX (GUID2),13,4)||'-
'||SUBSTR(RAWTOHEX (GUID2),17,4)||'-
'||SUBSTR(RAWTOHEX (GUID2),21,12))
```

Challenge “5”

Any obsolete dataset (small or large) has to be removed from the database

- Rows deletion from compressed already data takes considerable time and moreover generate considerable amount of undo and redo.

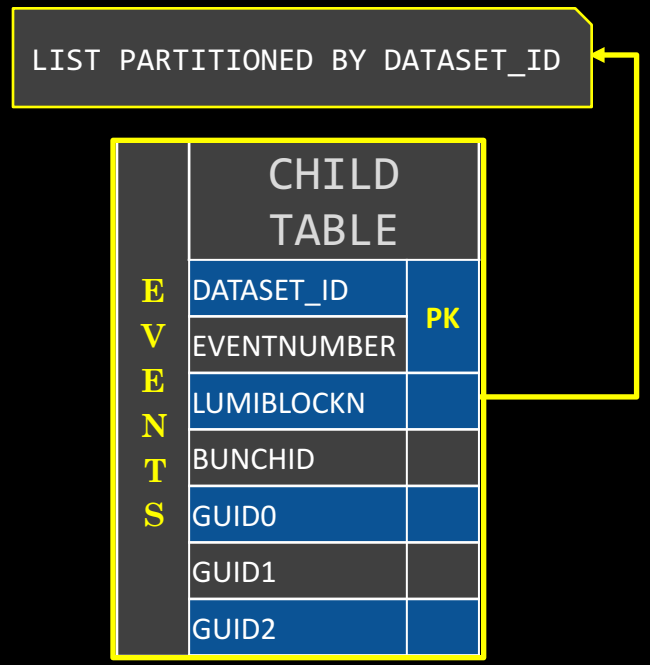
- Delete of 10s or 100s of millions rows takes hours or it may not succeed because of the needed significant undo.

Tested approach for challenge “5”

Idea: do not delete row by row, but scratch a complete dataset in a straightforward way: partition the “child table” in a way appropriate for simple partition removal.

Best approach: LIST partitioning

```
CREATE TABLE  child_table (
DATASET_ID  NUMBER(10,0),
...
CONSTRAINT cons_name PRIMARY KEY (..) using index COMPRESS 1
LOCAL
) pctfree 0  COMPRESS BASIC tablespace &&m_tbs
PARTITION BY LIST(DATASET_ID)
( PARTITION DATASET_ZERO VALUES(0) );
```



List partitioning: an operational challenge

Caution: List partitioning is appropriate, but is **an operational challenge** as partitions must be pre-created for each new partition key value.

- This is a burden as new datasets have to be registered any time in the system.

In-house created solution:

- Partition is automatically created for the staging and the final child tables whenever a new dataset (partition key value) is created into the "parent" table.
- "After insert" row level trigger fires and executes a PLSQL procedure responsible for `ALTER TABLE … ADD PARTITION`
- Each partition creation action is logged into a dedicated logging table

Partition removal setup

Setup:

- Staging table: as the nature of the data is transient, partitions are removed automatically on chosen interval via an Oracle scheduler job
- Child table: datasets with flag ‘obsolete’ are removed automatically by a scheduler job: relevant partition is dropped.

Result: The operation is very efficient as table and index partitions are removed without a need for expensive undo and redo.

Goal achieved: Obsolete data is automatically & efficiently removed.

Challenge “6”
Statistics gathering on table with billions of rows

- Gather statistics on very large table is time and resource consuming
- What level of statistics are enough for the use cases we have?

Stats gathering

Problem: The Oracle auto stats gathering is not appropriate for the partitioned "child table" with billions of rows because:

1. By default it gathers stats on partition level and on global level
2. It may decide to compute histograms for some of the columns
3. New stats may appear at any time

Solution: Better to have customized settings for the stats gathering on certain tables because :

- **DBAs and developers know best what and when has to be updated**
- **Partition level stats lead to undesired execution plan which changes when using bind variables.**

Remark: Seen behavior of CBO choosing full partition scan, because of tiny partition segment, instead of PK unique scan. Reusing such plan on partition with 100s millions of rows is a disaster.

Customized stats gathering

Setup:

1. Global table stats only

```
exec DBMS_STATS.SET_TABLE_PREFS(‘owner’, ‘table’, ‘GRANULARITY’, ‘GLOBAL’ );
```

2. Forbid histograms

```
exec DBMS_STATS.SET_TABLE_PREFS(‘owner’, ’table’, ‘METHOD_OPT’ , ‘FOR ALL COLUMNS SIZE 1’ );
```

3. Percent of the table content to be considered for computation

```
exec DBMS_STATS.SET_TABLE_PREFS(‘owner’, ’table’, ‘ESTIMATE_PERCENT’ , 1 );
```

4. Degree of parallelism in the stats gathering

```
exec DBMS_STATS.SET_TABLE_PREFS(’owner', ‘table', 'DEGREE', 2 );
```

5. Lock statistics

```
exec DBMS_STATS.LOCK_TABLE_STATS( ’owner’,‘table’);
```

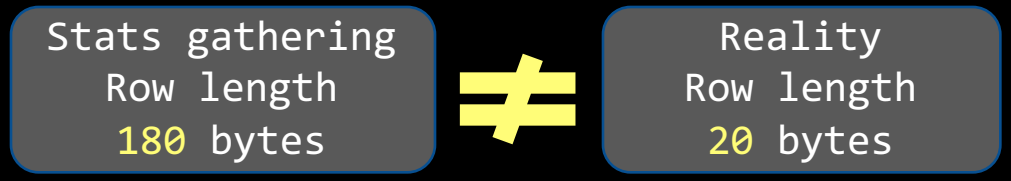
6. Gather statistics with scheduler job monthly with “force=true” option.

More about stats

Oracle computes stats on the virtual columns

Question: Virtual columns do not take space but AVG_COL_LEN is computed for them. Their AVG_COL_LEN is added to the row length AVG_ROW_LEN. Why?

Findings: In our case with the "child table" data, Oracle stats gathering comes up with AVG_ROW_LEN = 180 bytes (while in reality on average each row takes 20 bytes) because of the compression and "raw" data type columns and the virtual columns on top of them.



Challenge “7”
Performance of data retrieval
&
Queries execution plan stability

- Main use case: give me the GUIDs (file identifiers) for a list of protons collision events [1….thousands events] from list of LHC runs [1..tens or hundreds runs]

Note: Run number is part of the dataset name

Query with long IN list

Question: How to compose a SQL statement that could have hundreds of IN list values?

Caution:

1. Having long IN list with hundreds of literals makes difficult SQL statement parsing and execution plan stability is at risk.

```
SELECT GUID0, GUID1, GUID2
FROM  parent_table pt, child_table ct
WHERE pt.dataset_id = ct.dataset_id AND
(RUNNUMBER, EVENTNUMBER) IN
( (279685, 569724665) , (279685, 994915396),
(279685, 1058293500), (…, …), …);
```

2. Long IN list with bind variables is not good option as well

3. The length of the IN list is not known.

Instead of IN list … use temporary table

Idea: instead of composing long IN list with values, the client session inserts values into an Oracle temporary table, joins it with the other tables, gets the result and commits (in order to clean the TMP data).

```
CREATE GLOBAL TEMPORARY TABLE  tmp_table
(run_id NUMBER(10),
event_id NUMBER(10))
ON COMMIT DELETE ROWS;
```

Idea: simplify the JOIN between the three tables ("dataset parent table", "dataset child table", TMP table), by encapsulating the query into a read-only view object.

```
CREATE OR REPLACE VIEW ...
AS  SELECT …  FROM parent_table, child_table, tmp_table
WHERE ...
WITH READ ONLY;
```

Advantages of using view objects

1. Simplifies the application queries
2. Certain complexity is hidden for the client applications
3. Only the needed columns are exposed to the client applications
4. Instructions (hints) can be specified in the view definition. Such instructions are good for ensuring execution plan stability.

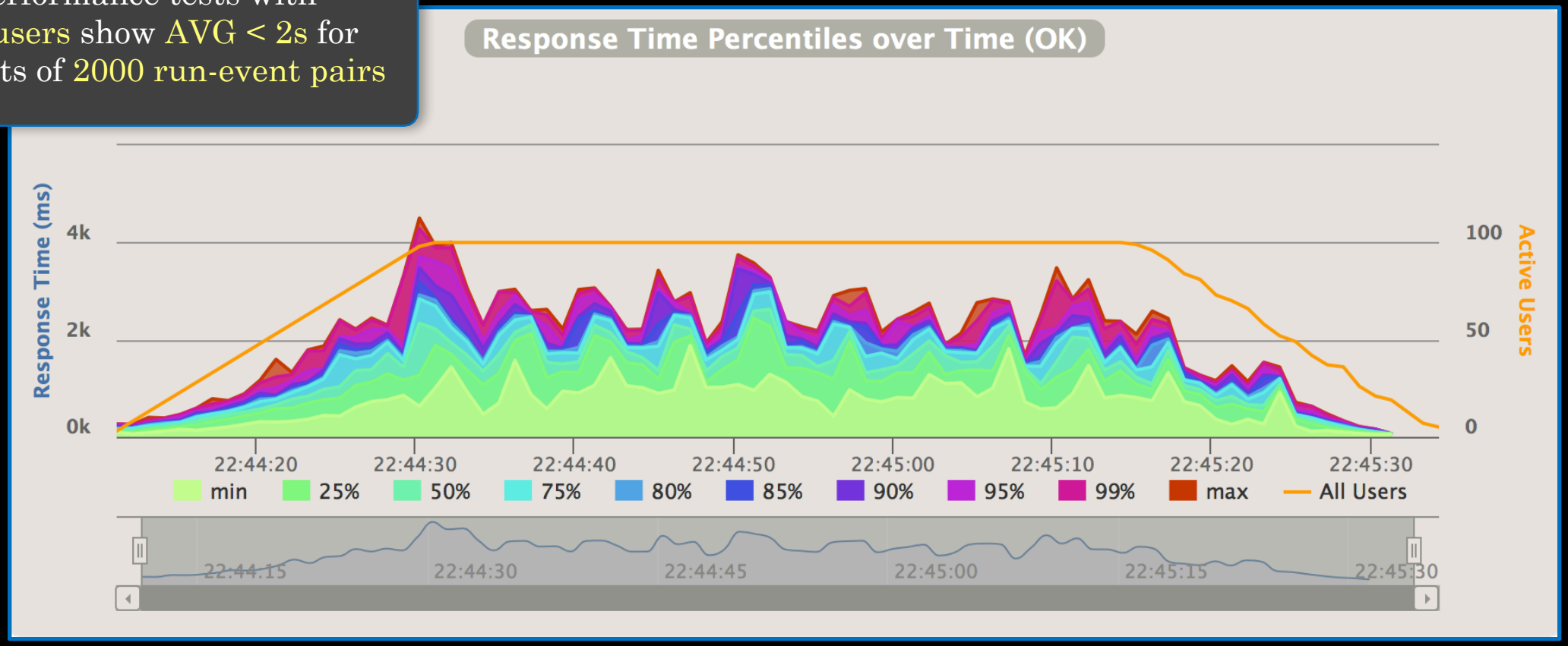
Example : We want Oracle to choose access path via index range scan, and we do not want index partition fast full scan.

```
INDEX_RS_ASC(table_alias(datasetID,eventID))  NO_INDEX_FFS(table_alias(datasetID, eventID))
```

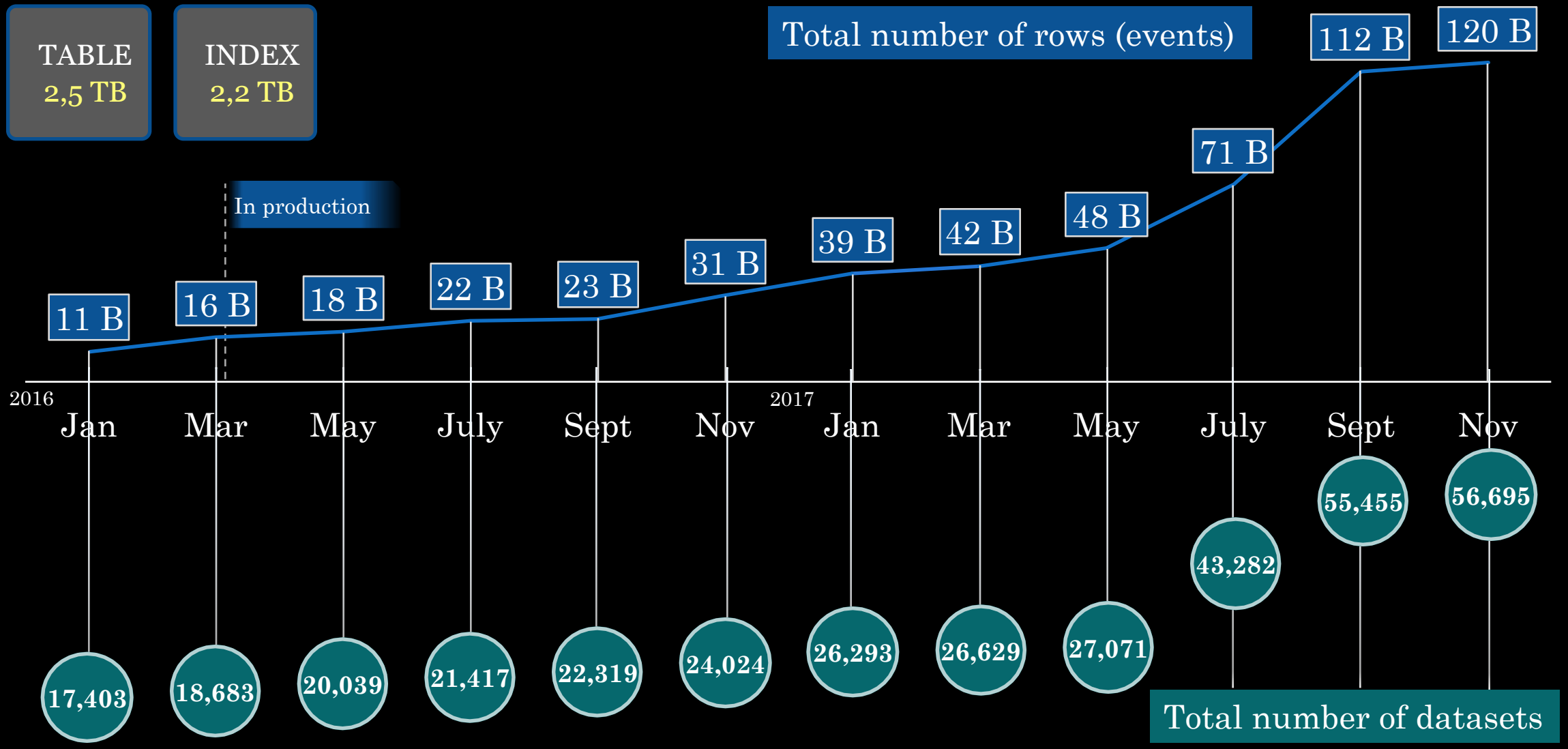
☑ Goal achieved: High performance and stable execution plans delivered

Data retrieval performance

Performance tests with
100 users show AVG < 2s for
requests of 2000 run-event pairs



System overview



Conclusions

- Information about the ATLAS particle collision events is well structured and stored in efficient way in ORACLE RDBMS
- Serious challenges are addressed in proper way
- The system is simple, robust and reliable
- The database handles seamlessly tens of billions rows per year

Thank you!

gancho@cern.ch, petya@cern.ch, elizabeth.gallas@physics.ox.ac.uk