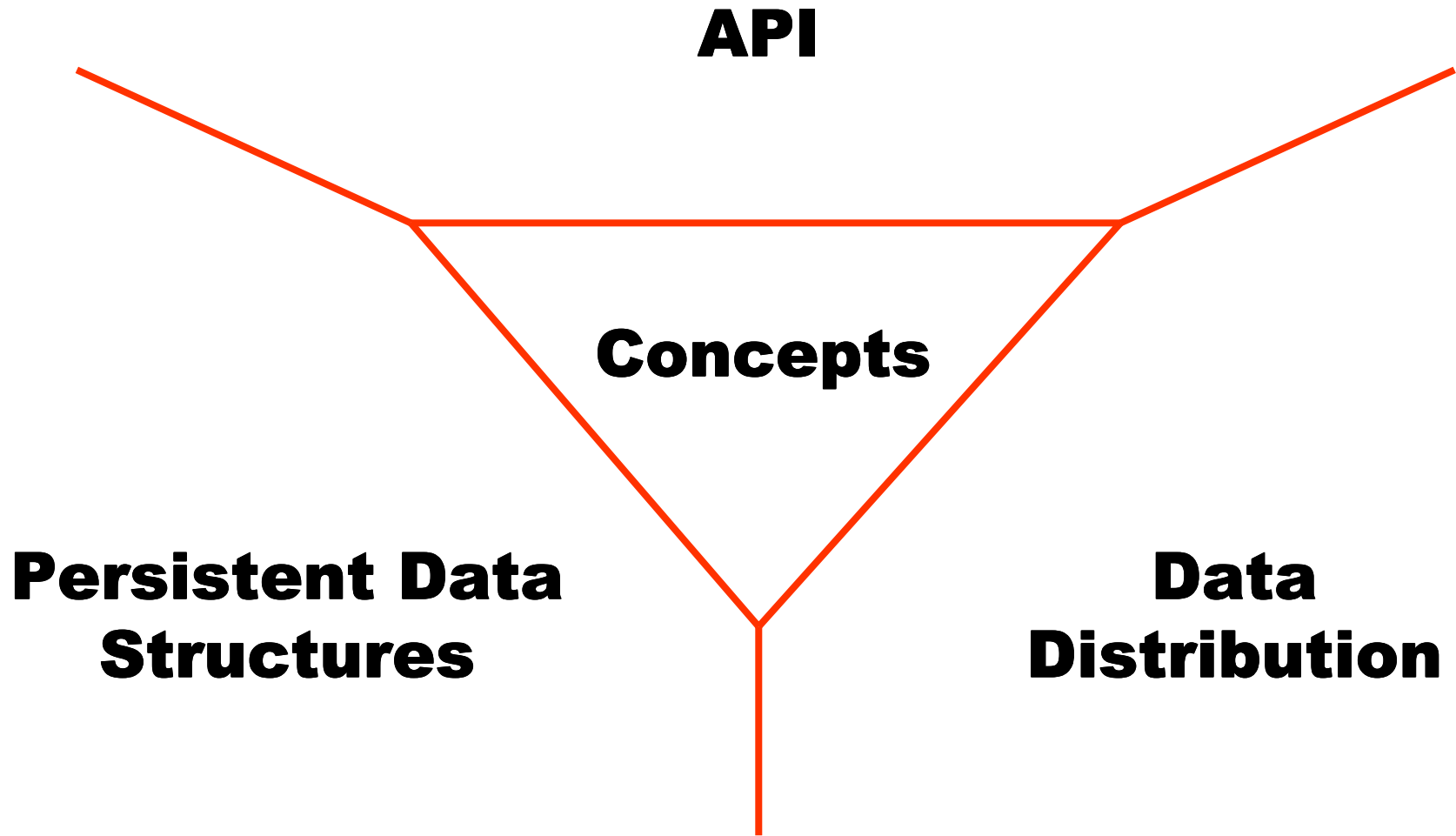


New Conditions/DB: Overview of Concepts and Design

Igor A.Gaponenko
Lawrence Berkeley National Laboratory

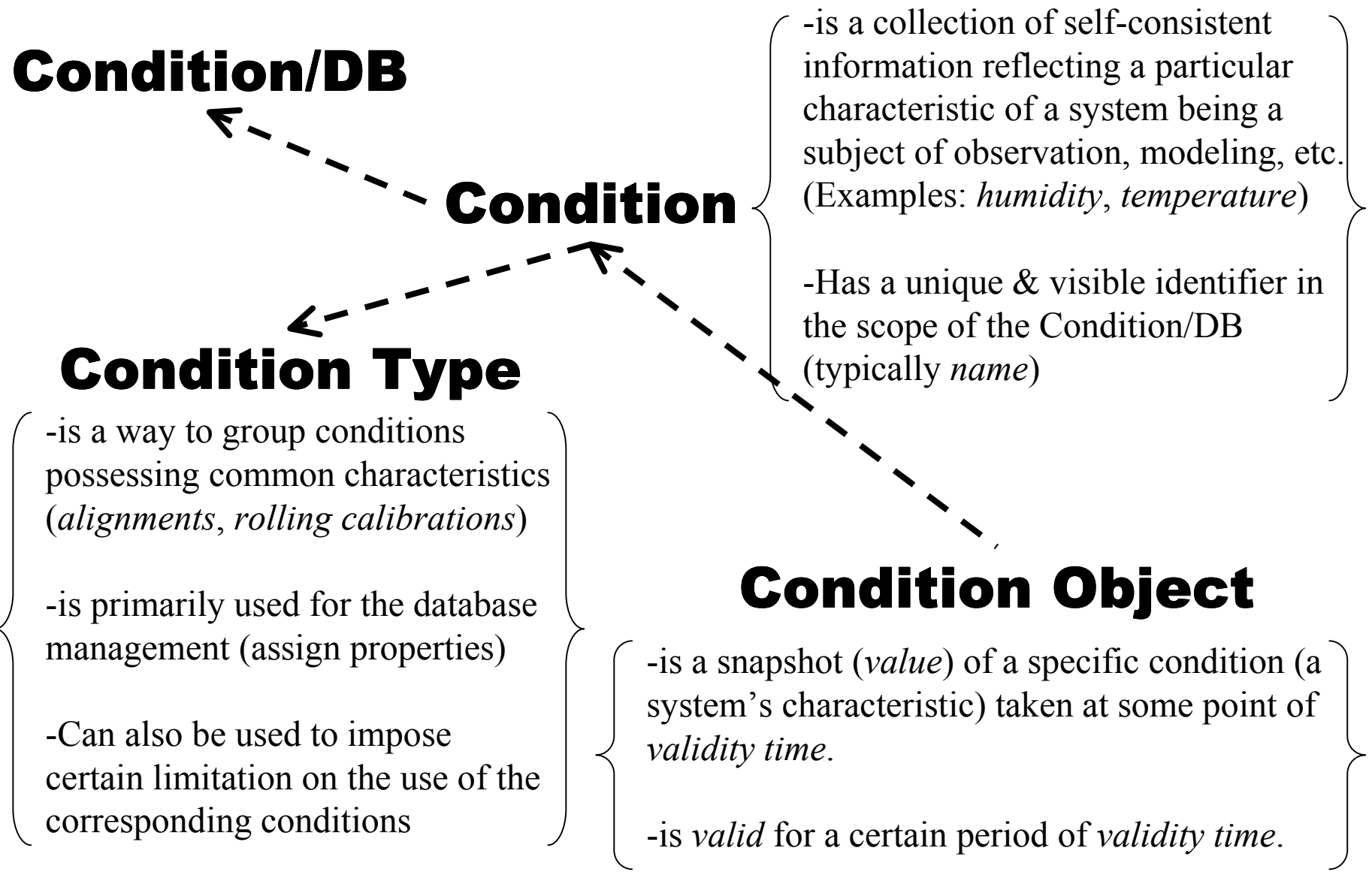
Topics covered by this talk



Why we need this mini-review?

- **Goals**
 - To present the audience a holistic picture of the new Condition/DB
 - To explain the most important design decisions
 - To get your feedback on the design to avoid:
 - Obvious (and not quite) mistakes in the design
 - Potential performance and scalability limitations
 - To hear fresh ideas you may have on the subjects covered by the talk:
 - It's never too late to improve the design(!)
- **Things not covered by the talk**
 - The design of the old Condition/DB
 - The criticism of the old Condition/DB

Concepts : About Terminology...



Concepts...

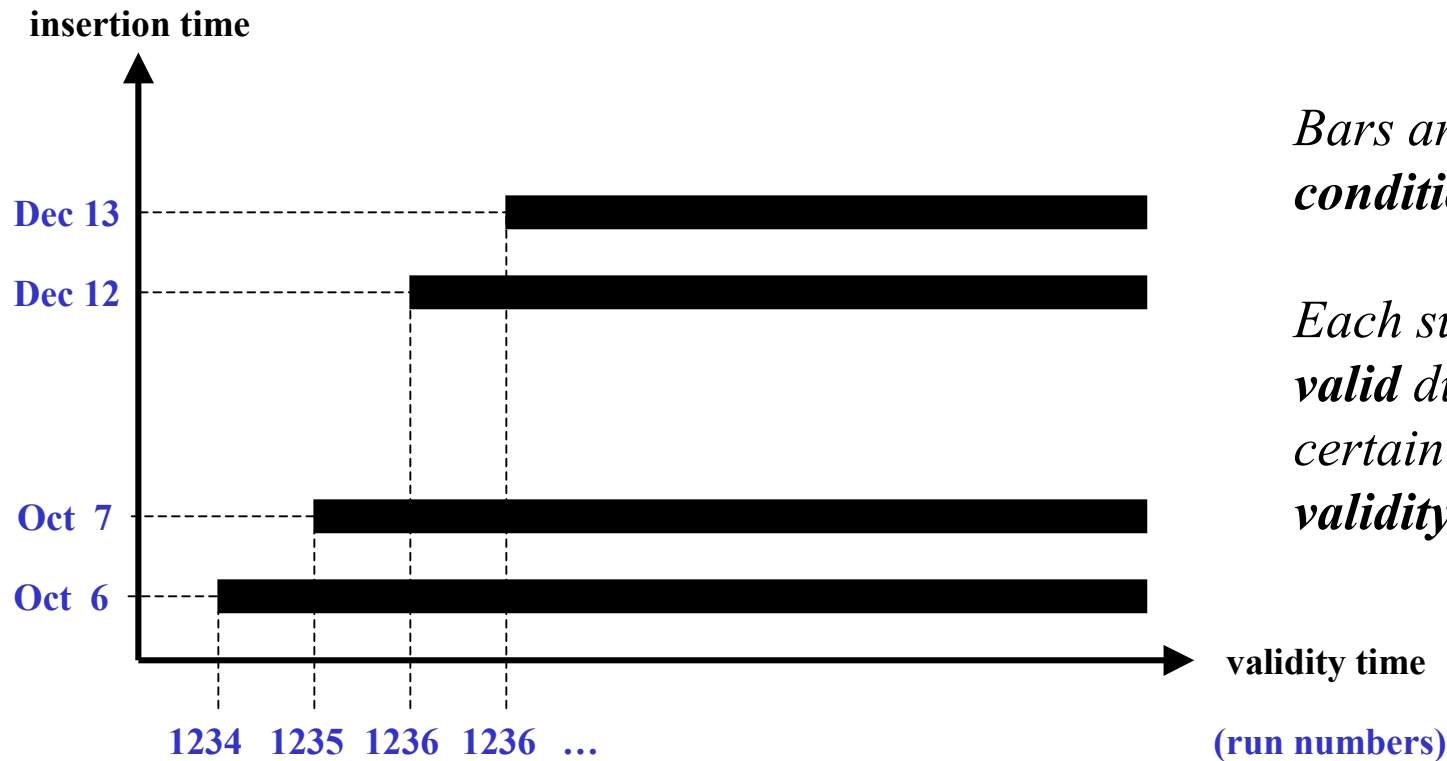
Conditions

Revisions

Partitions

Concepts : Logical Model of Conditions : storing

*Each condition lives in 2-D space of the **insertion** and the **validity** times*

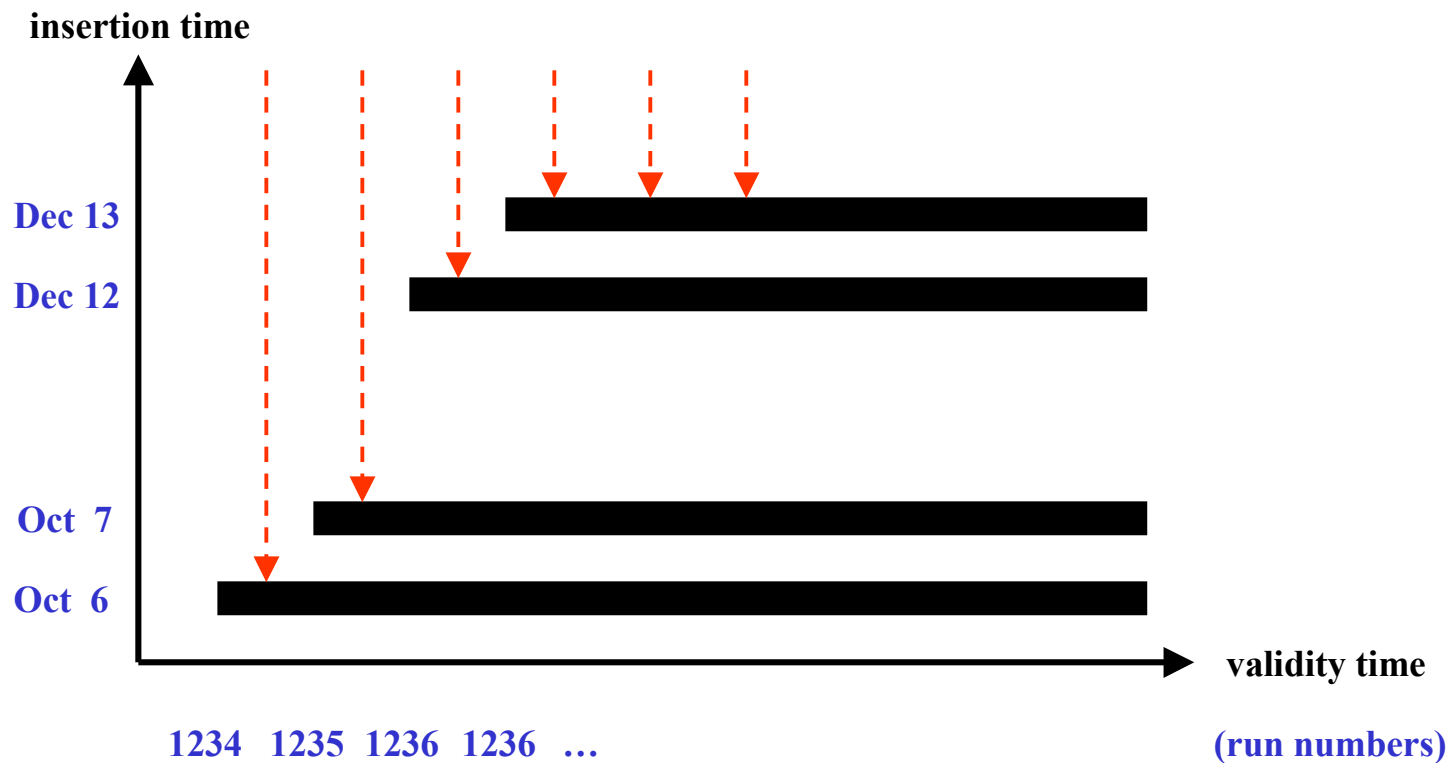


Bars are individual condition objects.

*Each such object is **valid** during certain interval of **validity time(line)***

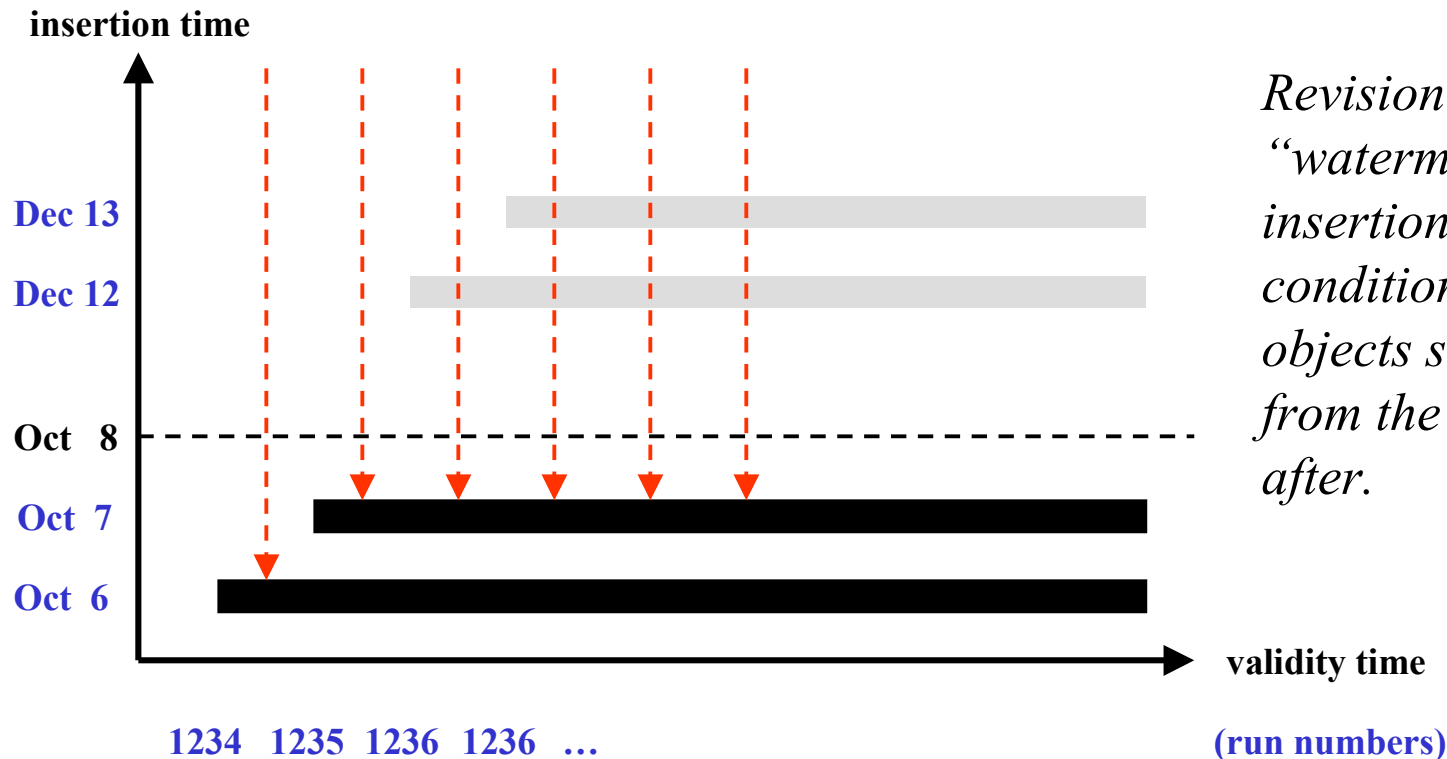
Concepts : Logical Model of Conditions : accessing

Over-simplified example: only the most recent conditions are read.



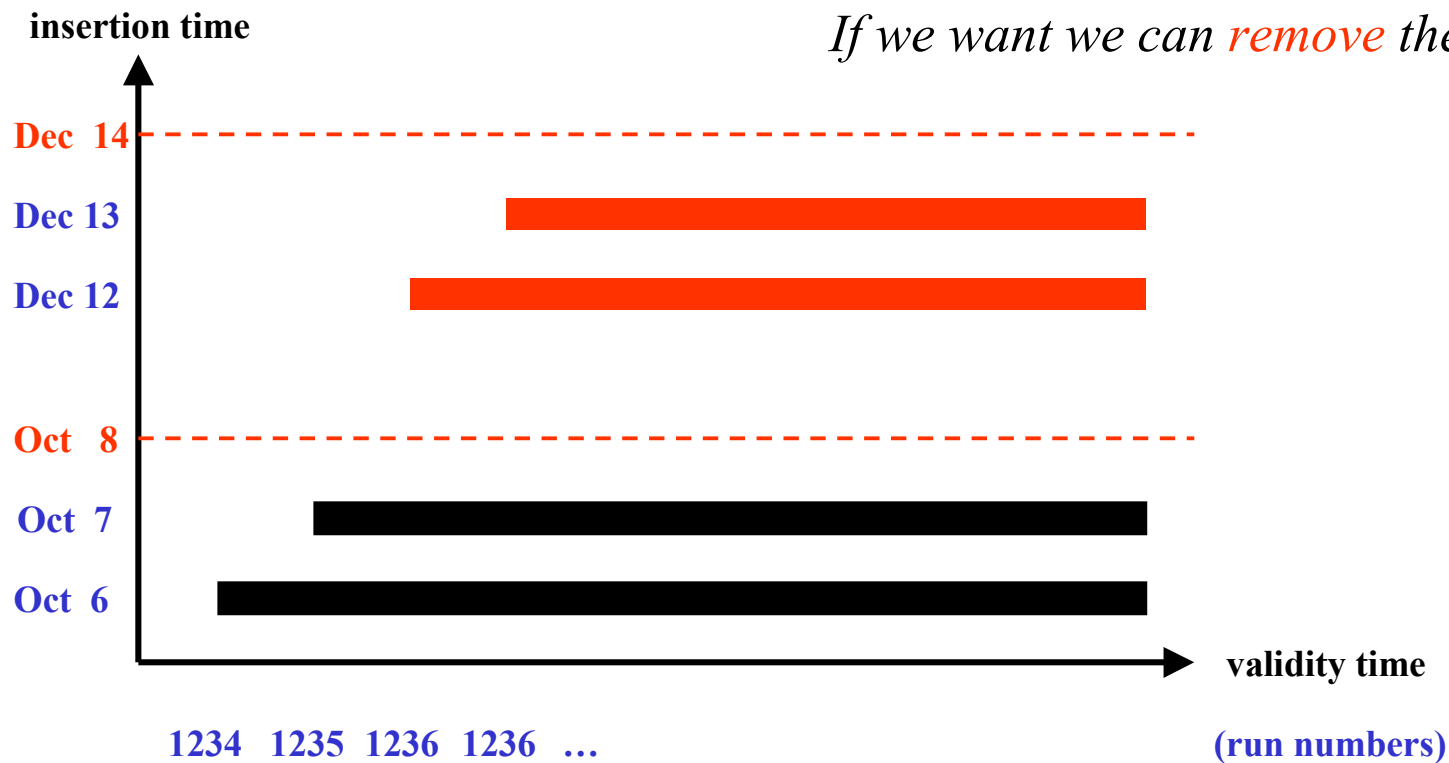
Concepts : Logical Model of Conditions : *revisions*

*Accessing conditions stored before
October 8. Ignore the newest ones.*

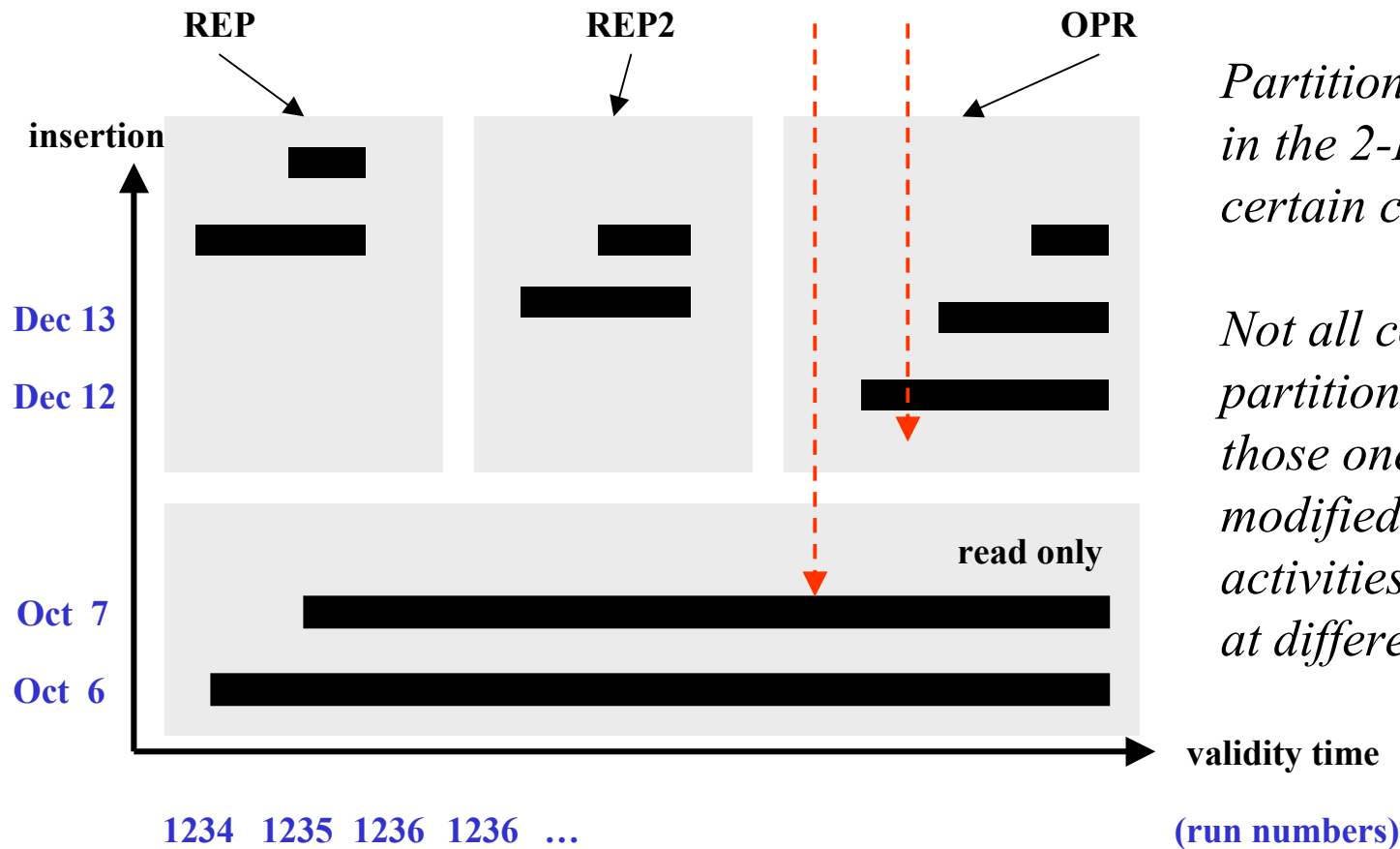


Concepts : Logical Model of Conditions : operations w/ revisions

*We can easily **calculate** which new objects were stored between them.
If we want we can **remove** them!!!*



Concepts : Logical Model of Conditions : partitioning

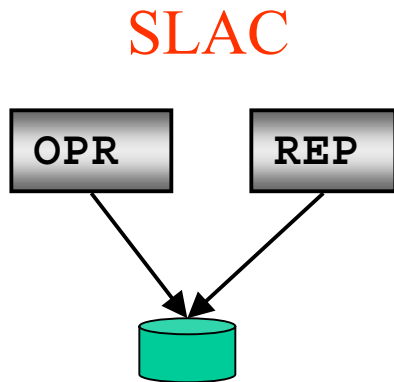


Partitions are boxes in the 2-D space of certain conditions.

Not all conditions are partitioned. Only those ones to be modified by different activities at a time or at different locations.

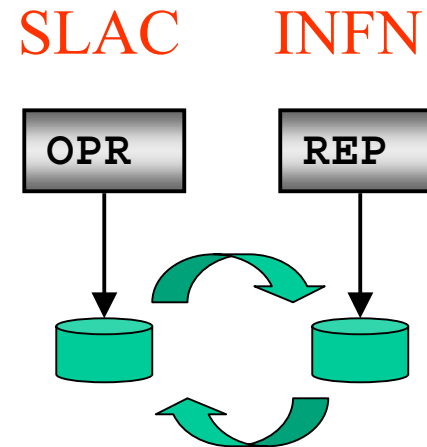
Concepts : Why do we need partitioning?

A: it's a unit of parallelism at persistent level



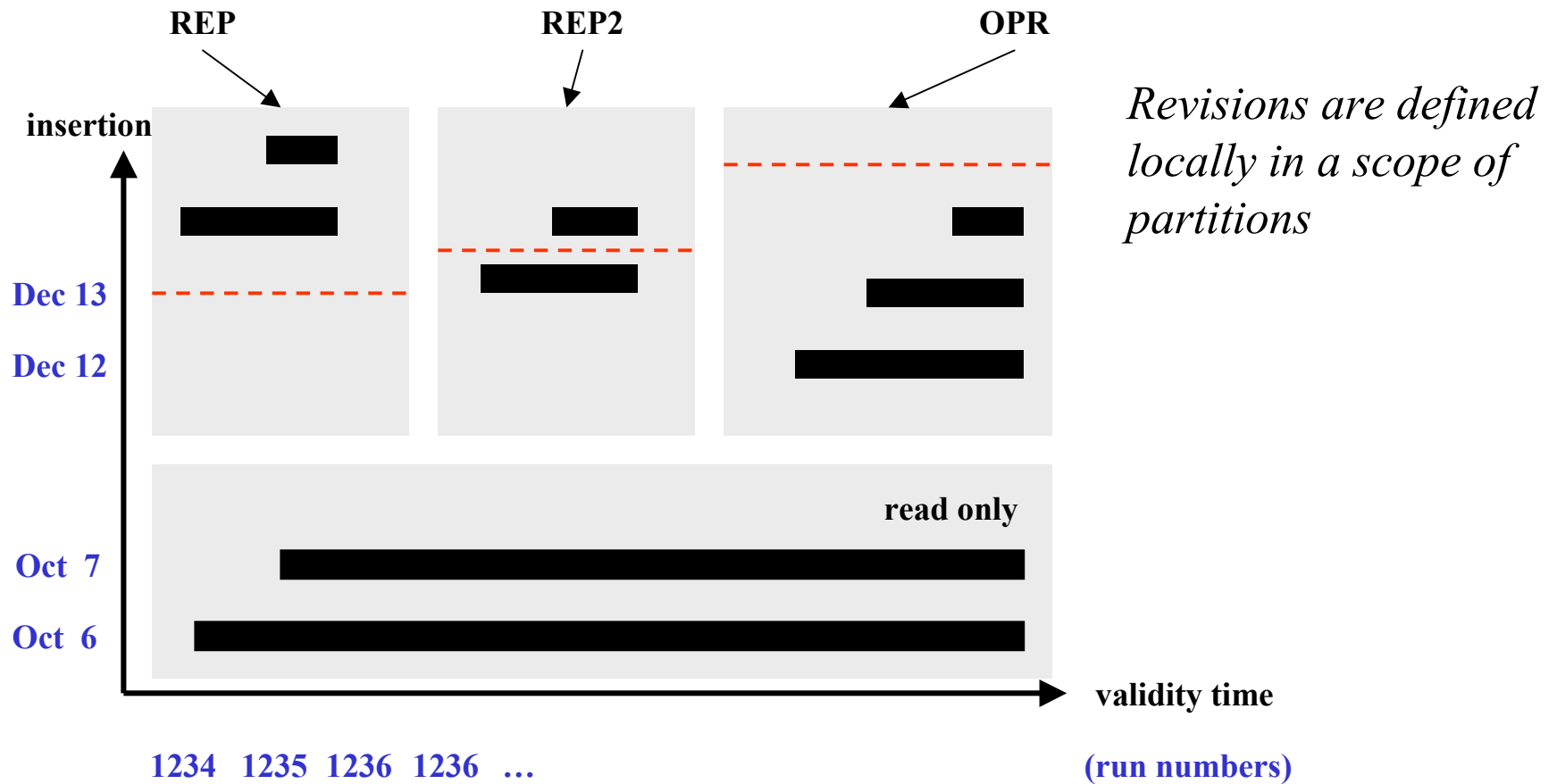
Shared Database

B: it's a unit of database management



Distributed Database

Concepts : Logical Model of Conditions : partitions & revisions



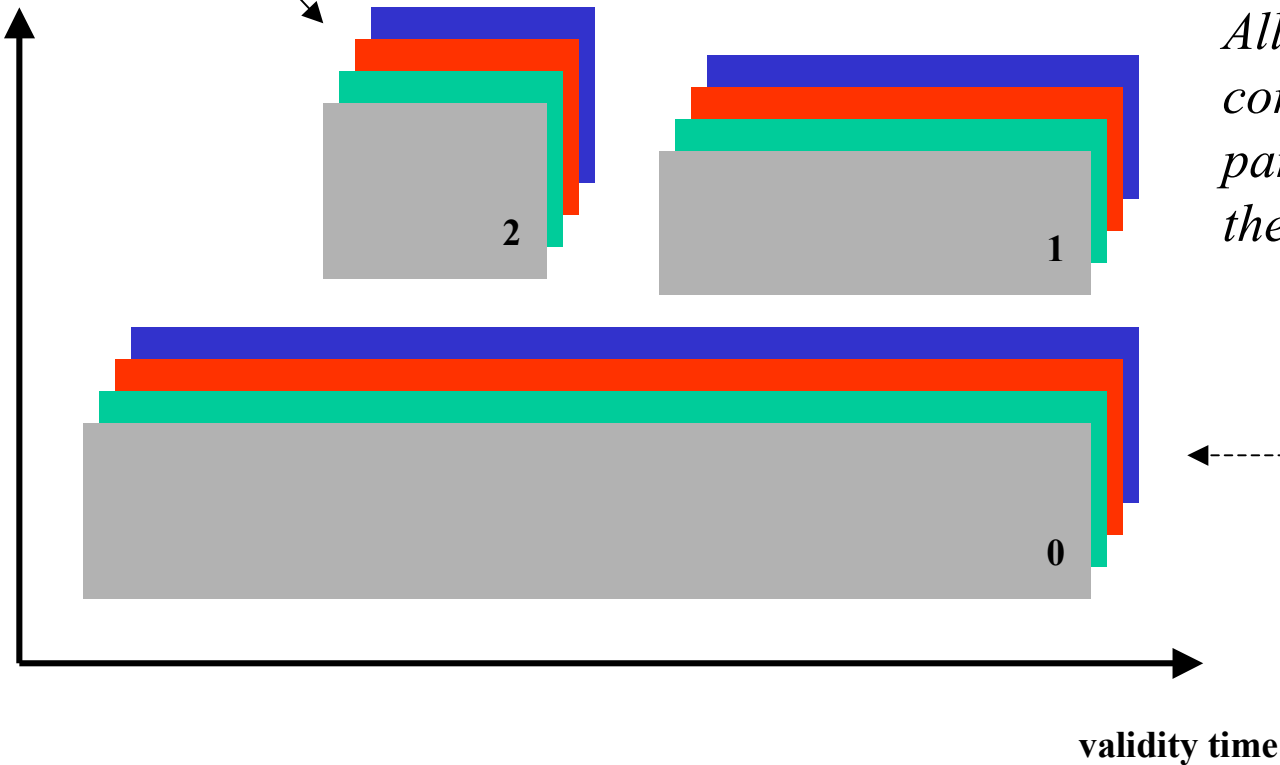
Partitions Layout

Concepts : Partitions layout : definition

Each color represents a condition

There is just one partitions layout in a database.

insertion time



All partitioned conditions are partitioned in the same way.

Concepts : Partitions layout : Partition modification time

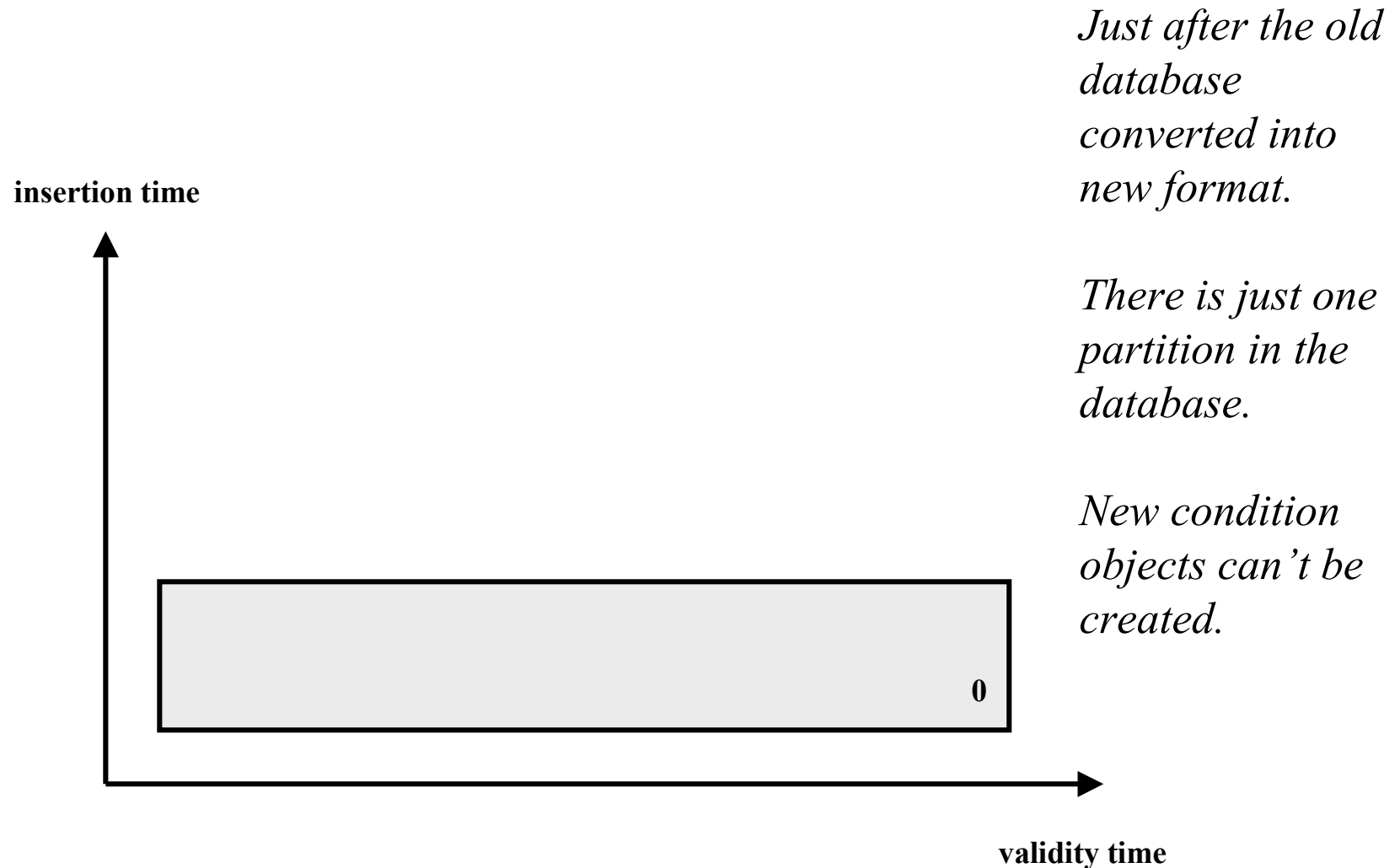
Each partition “remembers” when was the last (insertion) time any of its conditions was modified.



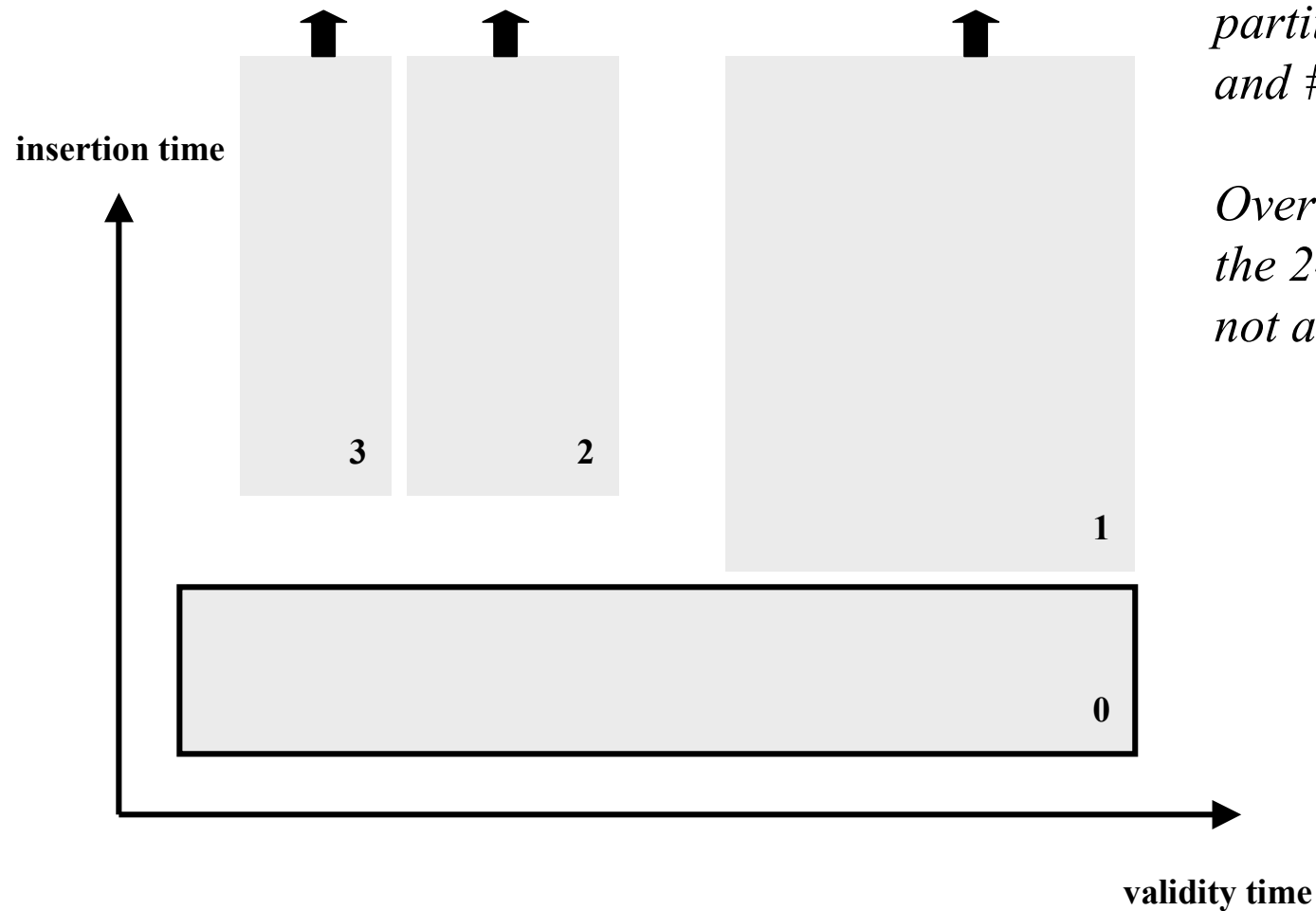
*This knowledge is used to identify the unique state of the database (see **State Identifier** later)*

NOTE: *There is important difference between the current time and the partition modification time!!!*

Concepts : Partitions layout : example (1)



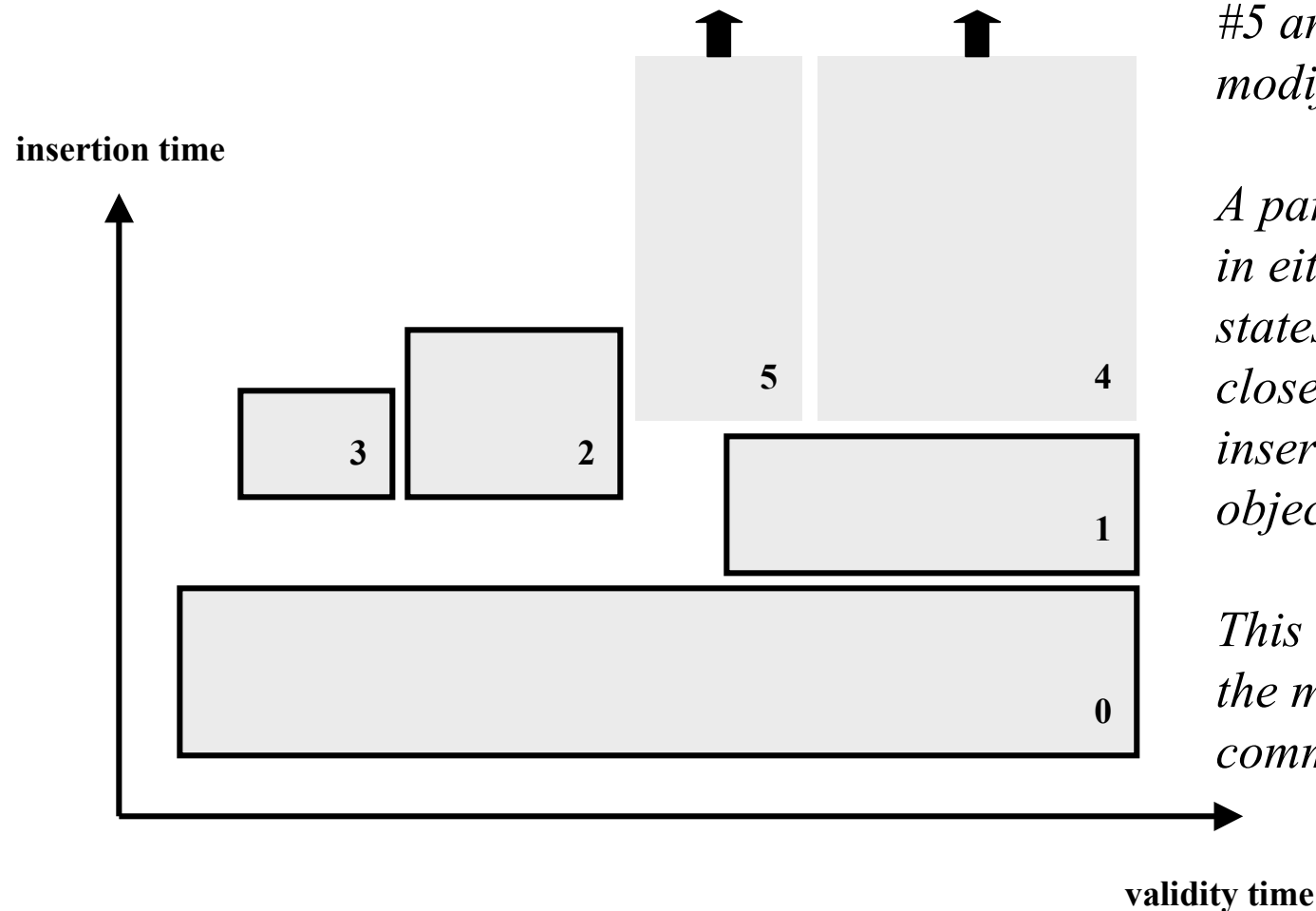
Concepts : Partitions layout : example (2)



Three open partitions #1, #2 and #3 created

Overlapping in the 2-D space are not allowed.

Concepts : Partitions layout : example (3)

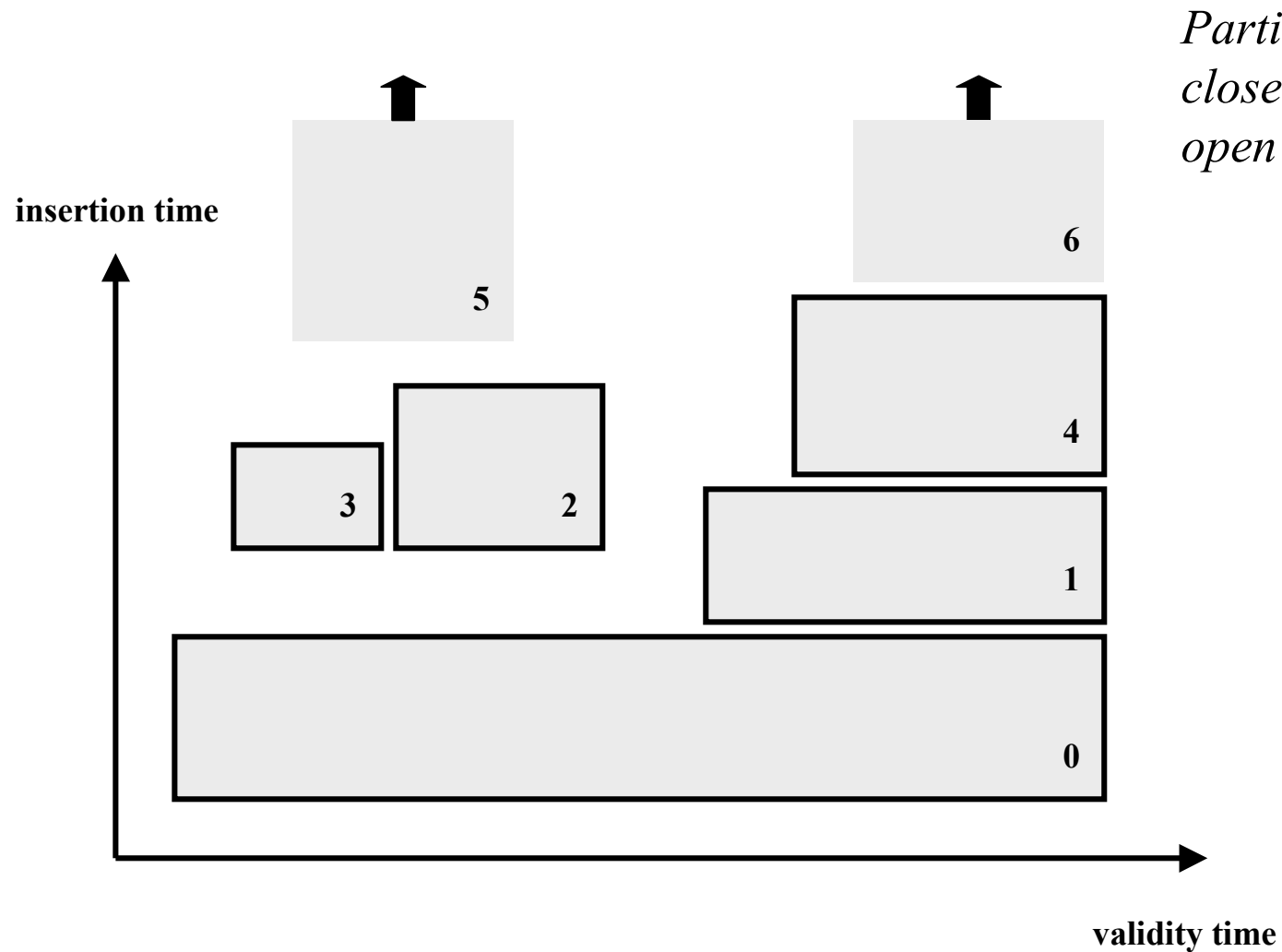


Partitions #4 and #5 are open for modifications

A partition can be in either of two states: open or closed (for insertion of new objects).

This handled by the management commands.

Concepts : Partitions layout : example (4)



Partitions #4 was closed and #6 was open instead

Concepts : Open questions about partitions...

*What about newly created conditions?
Do we need to create empty partitions
for them “back” in the insertion time?*

See answer in the “Workbook”

*Does it make any sense to “join”
partitions?*

YES

*Minimizes the number of database
files, however it complicates the
Data Distribution when
performing cross-synchronization
in a distributed database setup.*

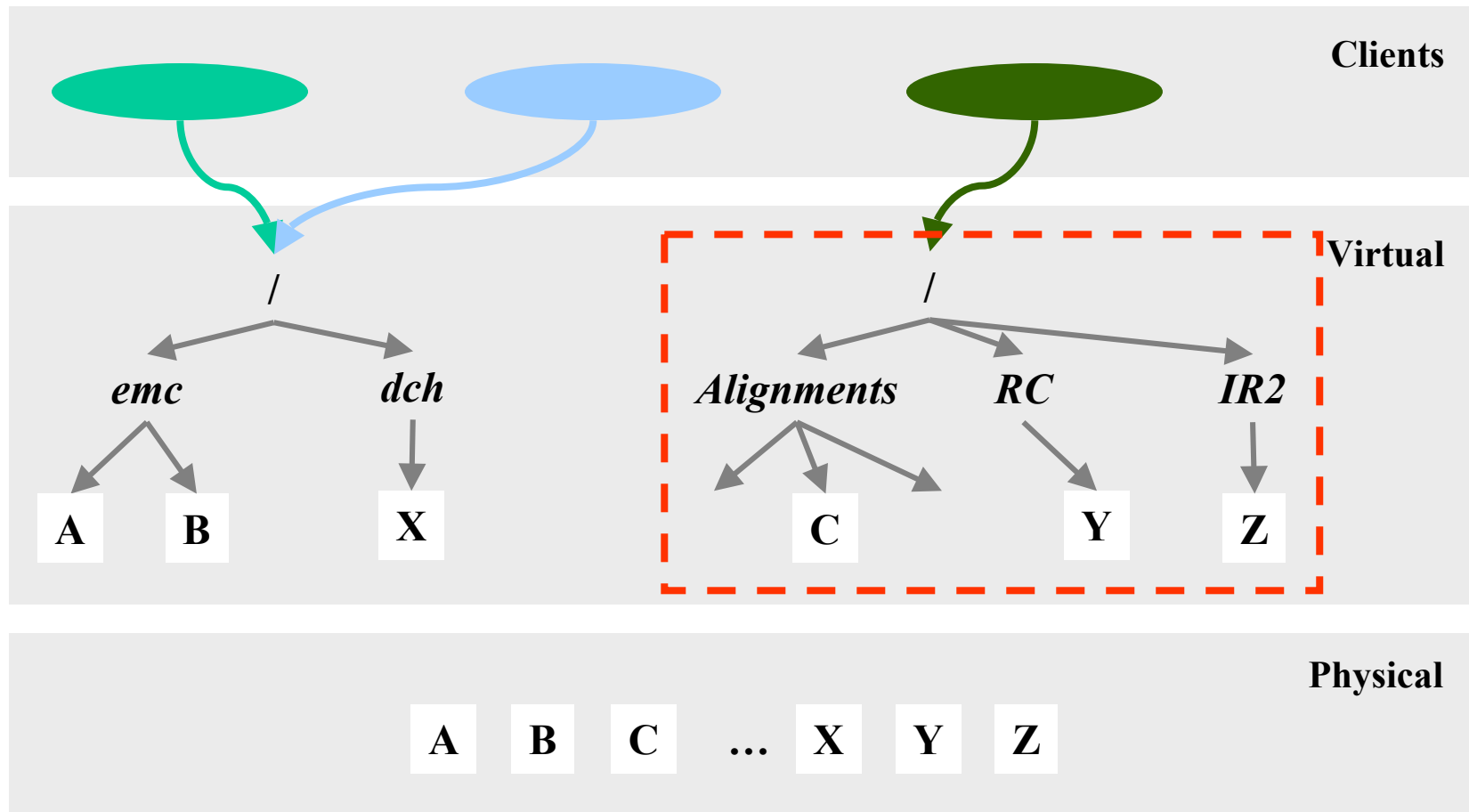
**Namespace of
Conditions**

Views

Folders

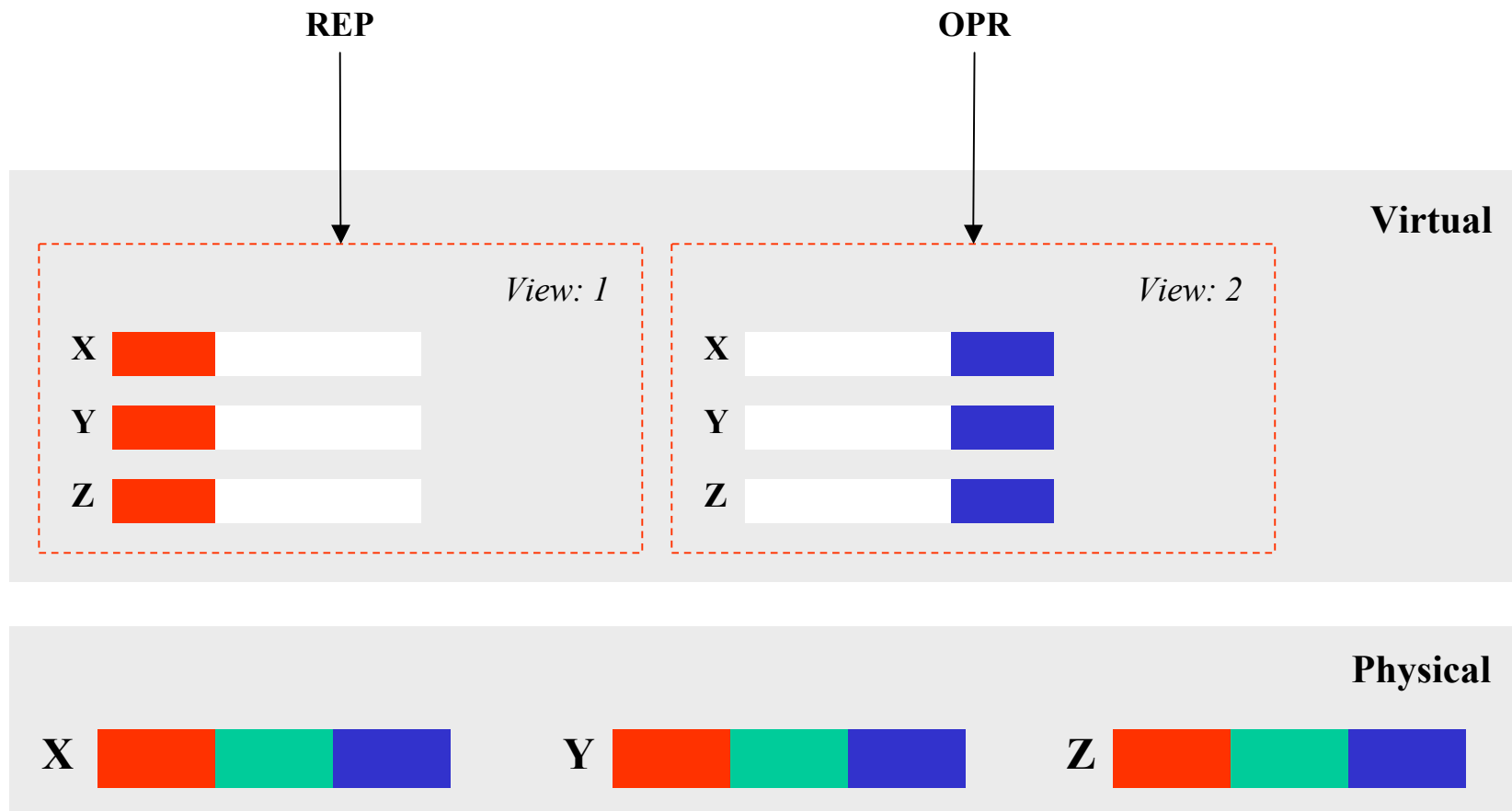
Concepts : Virtual Namespace of Conditions : *Views & Folders*

A concept of **views** lets us to treat the database contents in different ways for various kinds of clients.



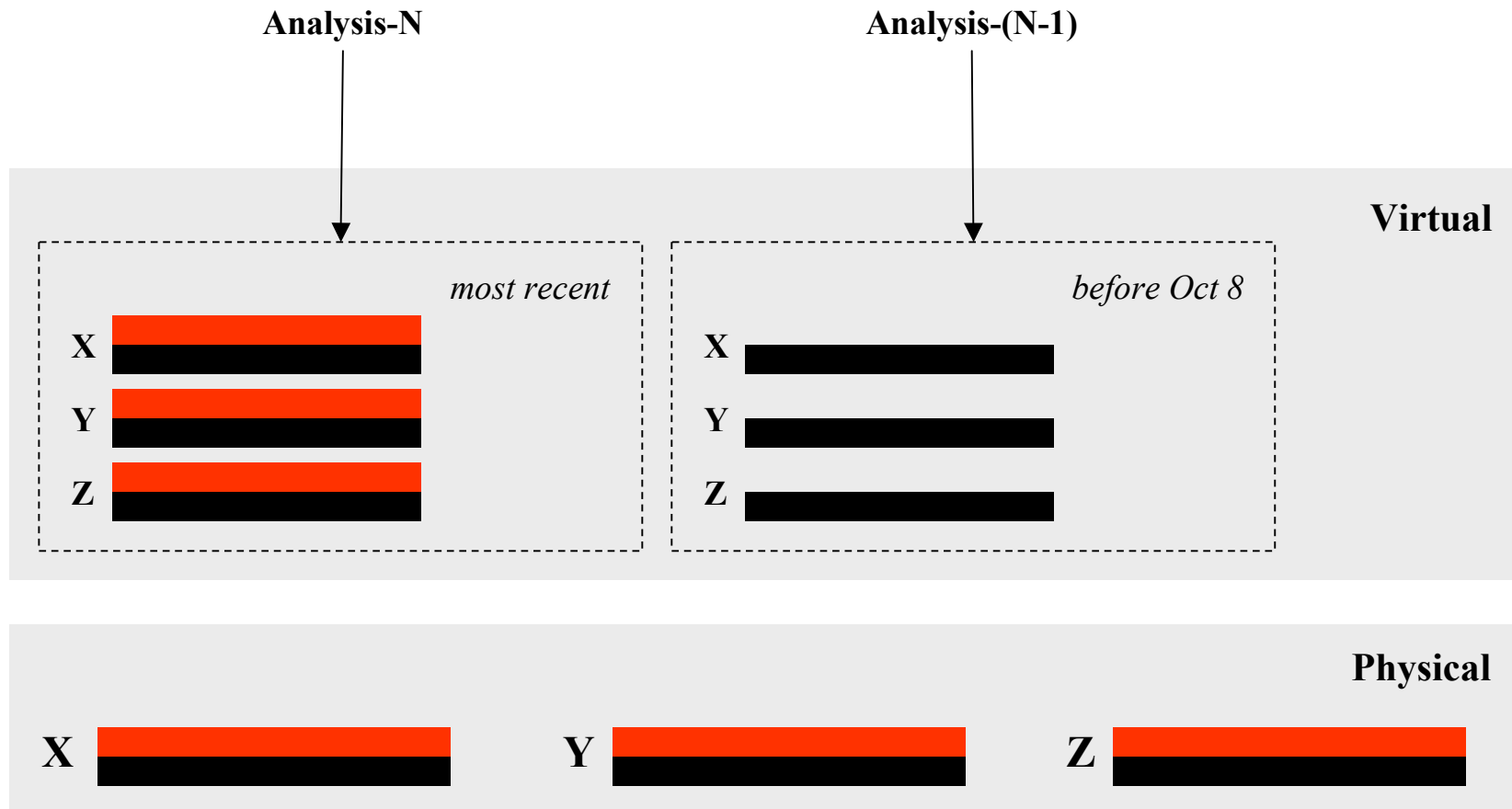
Concepts : Virtual Namespace of Conditions: *Views & Partitions*

Views may also restricts the ways the database contents is used by some clients.



Concepts : Virtual Namespace of Conditions : *Views & Revisions*

*Views may also be extremely useful to access the right slice of the database's contents (**revisions** in the current terminology).*



State Identifier

Concepts : State Identifier : the main idea

State Identifier is a short (a few bytes) description of the database content's state:

-its current value can be obtained through the API for specified validity time (will be explained later).

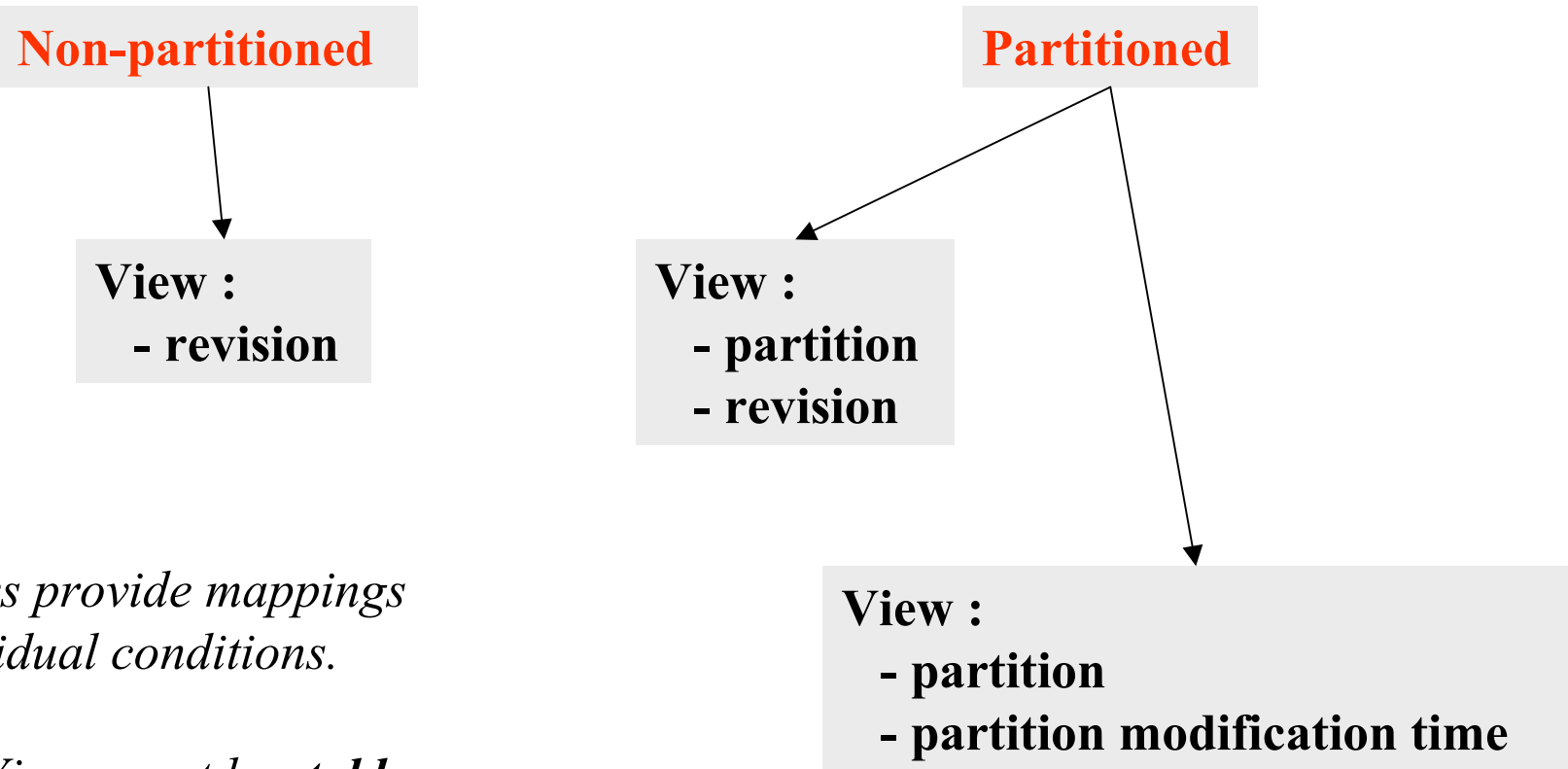
-it can be stored persistently (in event headers)

-it can be used to recover the same state of the database at the time of its (identifier's) creation.

This idea was originally suggested by Anders Ryd

Concepts : State Identifier : the implementation (1)

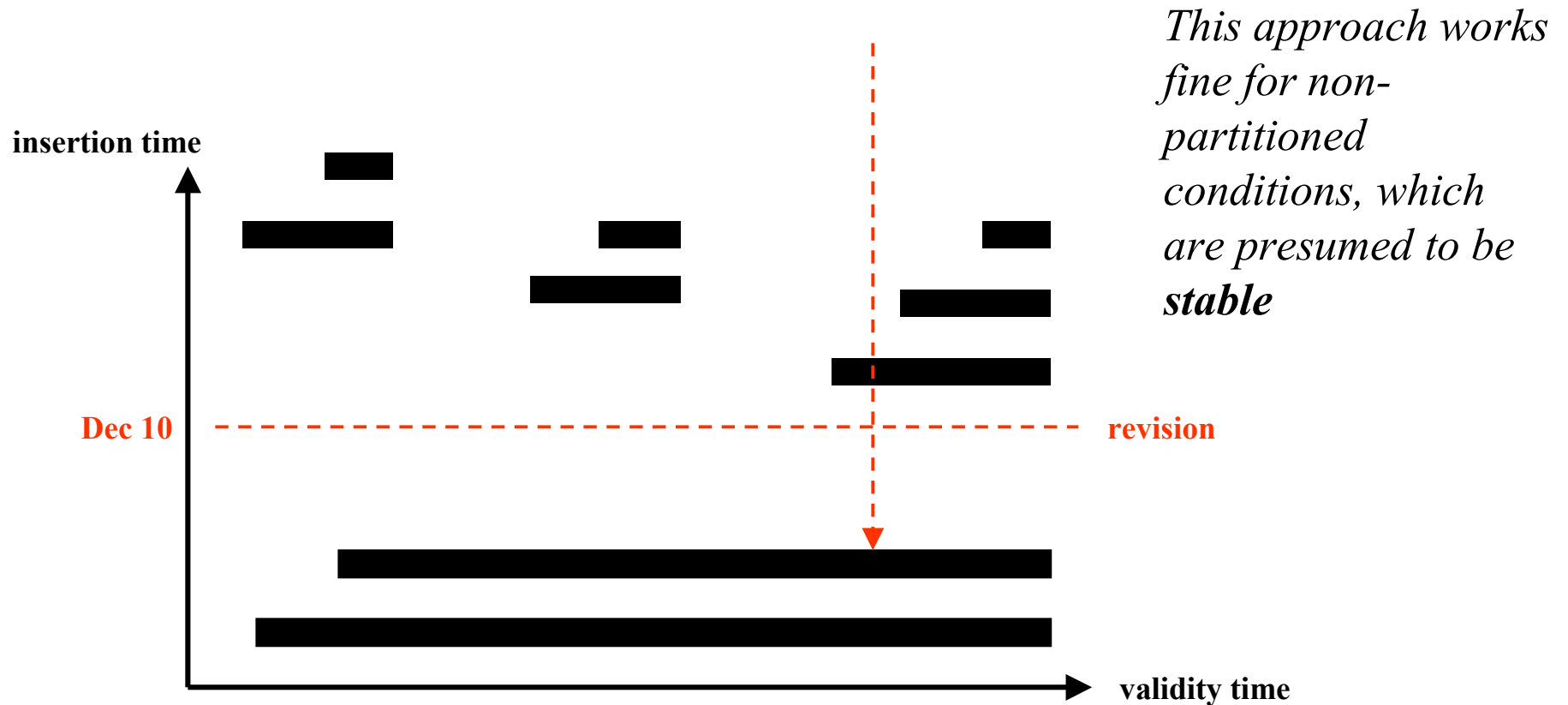
The definition of the “database content’s state” varies for partitioned and non-partitioned conditions:



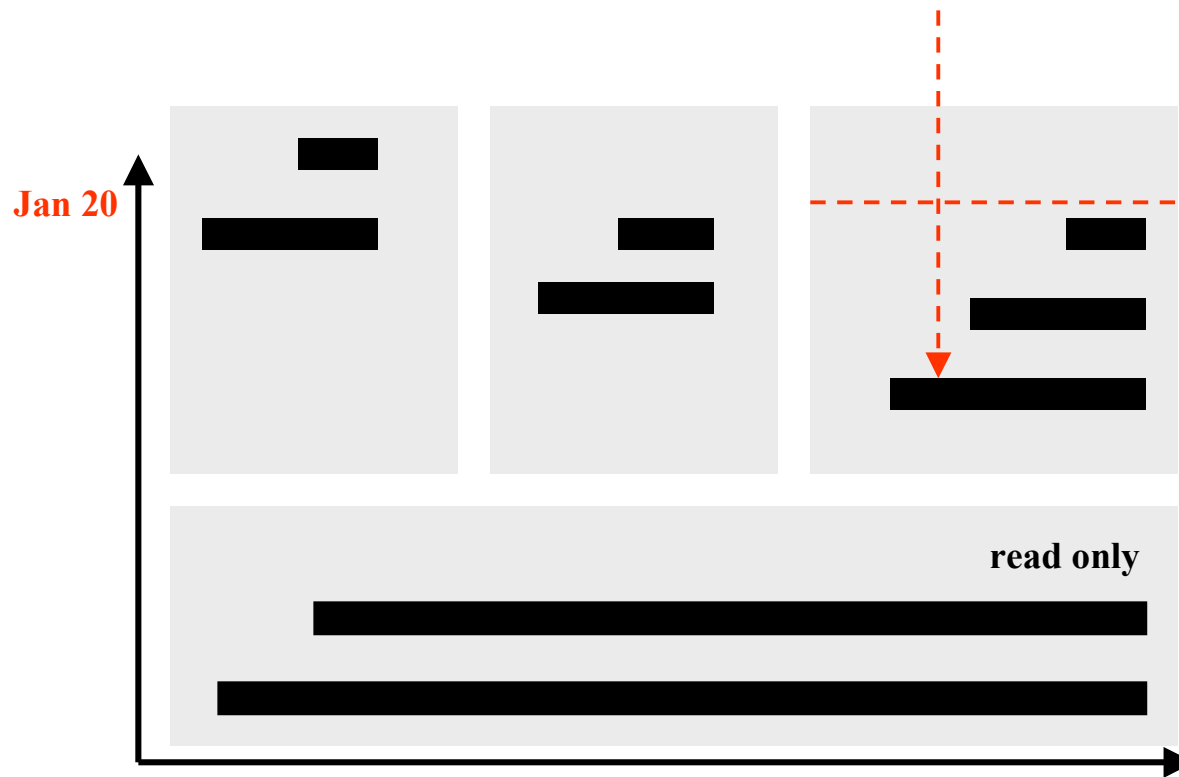
The views provide mappings for individual conditions.

*=> Views must be **stable***

Concepts : State Identifier : the implementation (2)



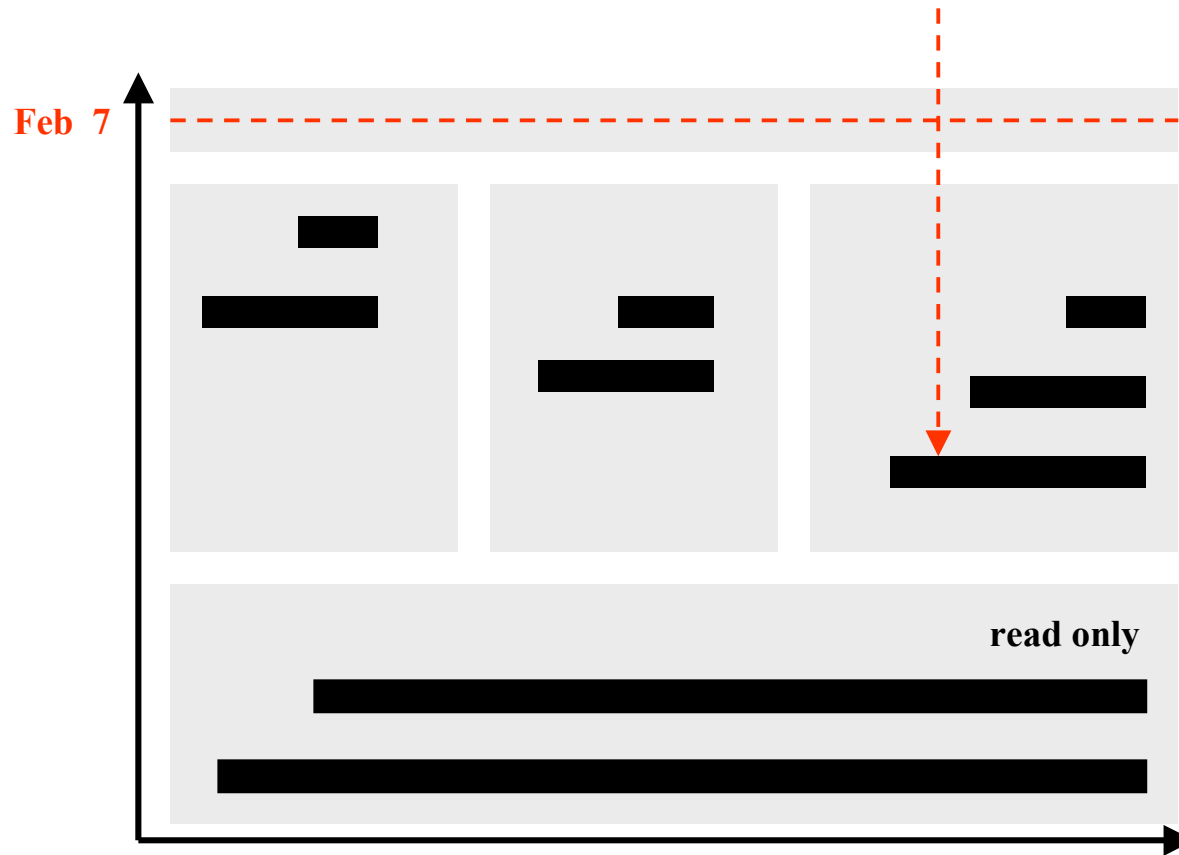
Concepts : State Identifier : the implementation (3)



Revisions can also be created in partitioned conditions.

The only difference from non-partitioned ones is restricted scope of these revisions.

Concepts : State Identifier : the implementation (3.1)

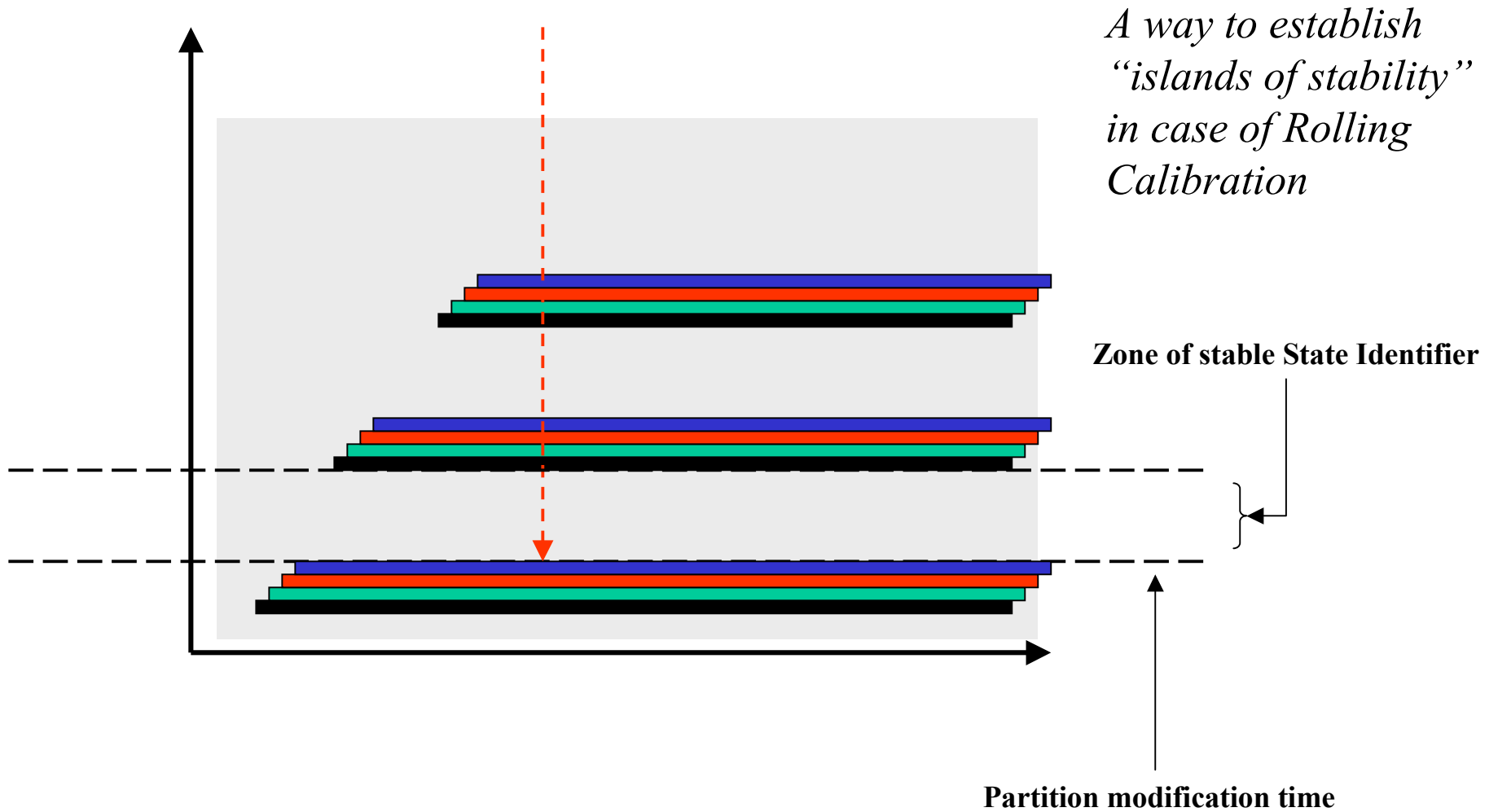


A fancy variation of the previous example.

We create an empty partition covering the whole validity timeline just to have a single revision.

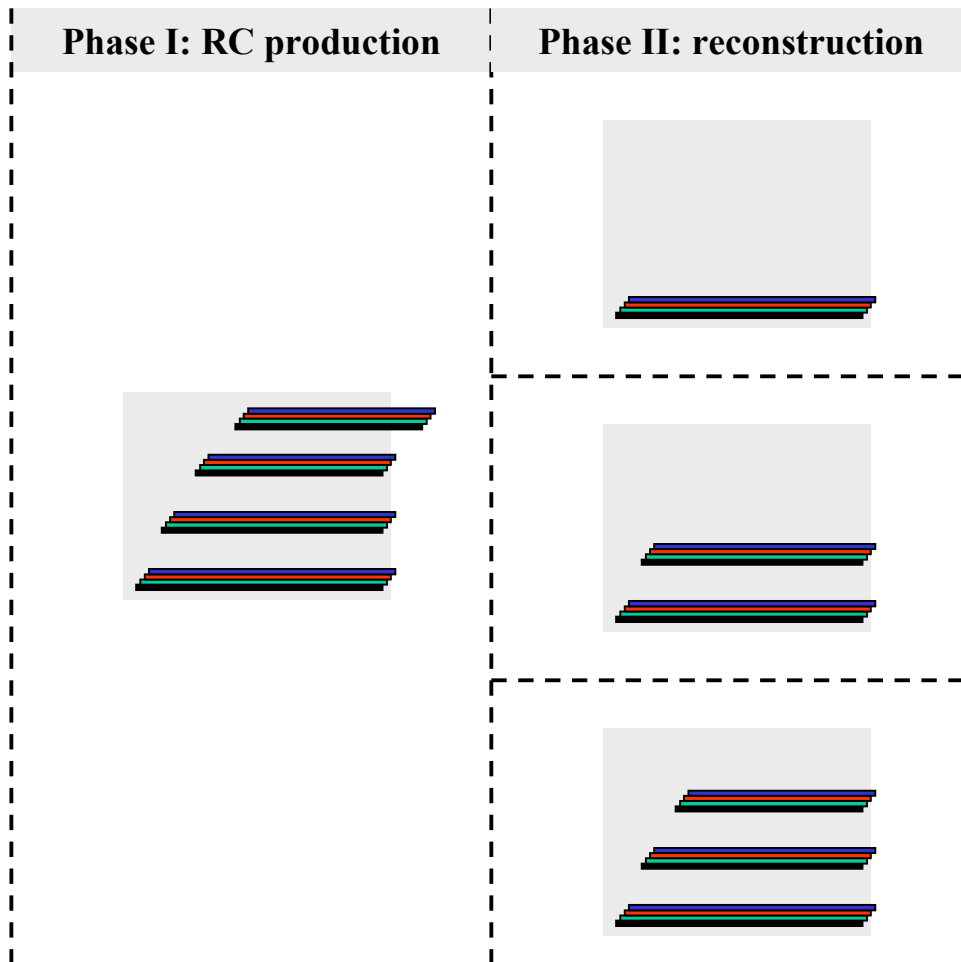
Why not?

Concepts : State Identifier : the implementation (4)



Concepts : State Identifier : use cases

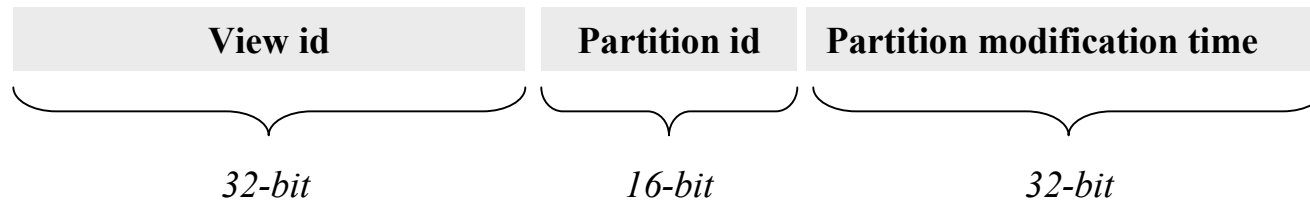
Rolling Calibration in 2-staged OPR :



Each group of nodes at the second stage runs against a snapshot of the Condition/Db generated at previous stage.

The values of the State Id are different for each of these stages

Concepts : State Identifier : the data structure



This works well for both partitioned and non-partitioned conditions.

The only difference is that partition information is not used for non-partitioned conditions since there is enough information (revision) for them in the corresponding view.

API...

Applications Programming Interface (C++)

API : (1)

- **Fundamental problem #1**
 - How to represent the variety of user defined types in the Condition/DB?
- **An ideal API**
 - Would automatically convert transient types into persistent implementations
 - Which is not possible for the following reasons:
 - (A) We already have a legacy classes (persistent, transient and proxies)
 - (B) There is the famous “schema evolution” problem.
 - (C) The programming language does not support it
- **This API (a compromise solution)**
 - Restricted support for the user defined types of conditions objects
 - At the same level the old Condition/DB does
 - But handling (creation & retrieval) of these objects is enhanced
 - Still separates meta-data from user-defined types
 - Hides underlying persistent technology for meta-data
- **Benefit of this approach**
 - Easy migration from the old API to the new one

API : (2)

- **Other features**
 - Use of *counted smart references* for API's interfaces
 - Data Clustering and Placement:
 - Users are not able to control the location of their objects in the database. The right value of the *clustering hint* is provided by the API.
 - Users' code is **forced** to create their specific persistent objects at a location suggested by the given value of the clustering hint.
 - Than API verifies if a new object is created in the right place.
 - Support for multiple (persistent) *technologies* and *implementations* (based on those technologies):
 - There is just one currently supported technology:
 - “**Bdb**”
 - » Using Objectivity as a persistent store
 - » User defined classes derive from **BdbCond/BdbObject** class
 - There are two implementation of the “**Bdb**” technology:
 - “**Wrapper**”
 - » Wraps the old Condition/DB into new API
 - » Restricted implementation of the API
 - “**Shared**”
 - » New persistent meta-data
 - » Different data placement and clustering for users' persistent objects

API : Data Clustering and Placement (1)

User-driven approach (old Condition/DB) :

Step A : User asks for a clustering hint

Step B : User creates an object as location suggested by the hint

Step C : User asks the Condition/DB API to register an object in meta-data

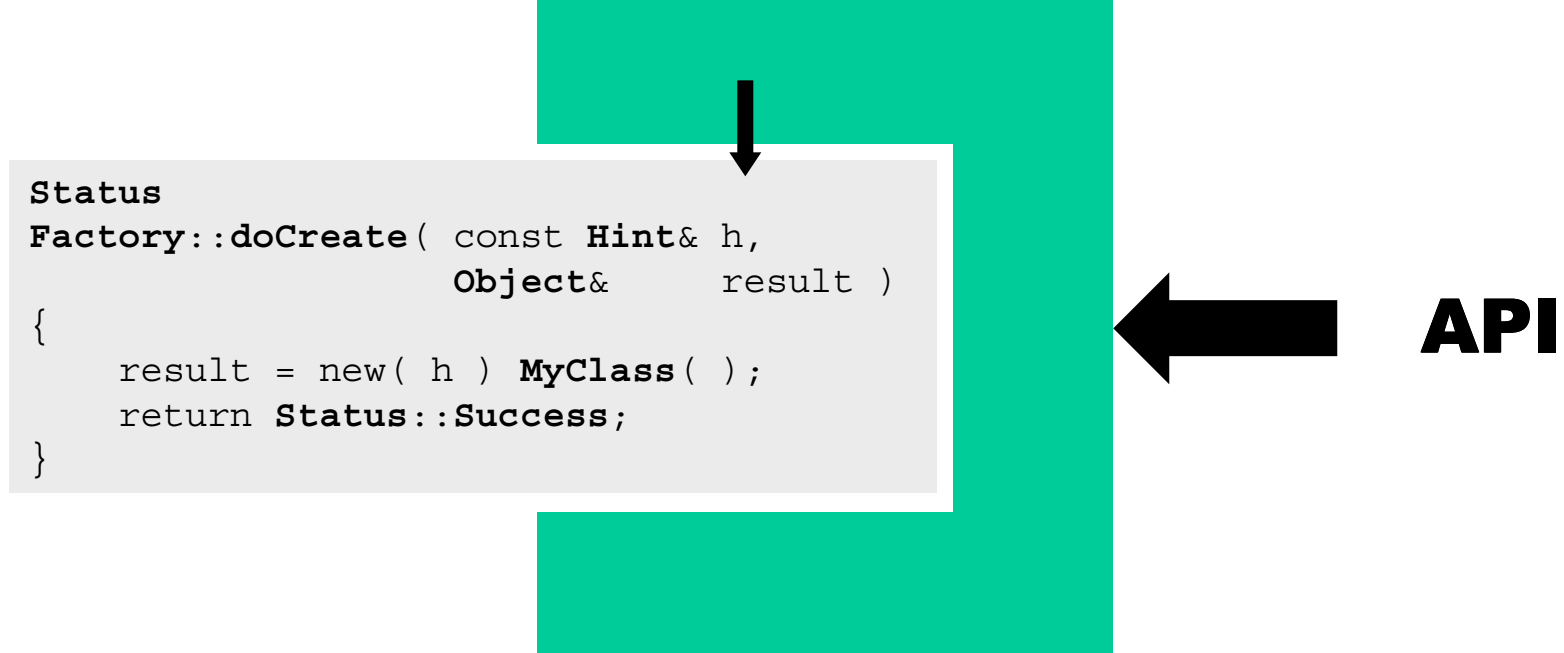
API-driven approach (new Condition/DB) :

Step A : User provides a piece of specific persistent object creation code to be executed by API. In practice it means creating a “factory” class by deriving from a special abstract class and implementing its **doCreate(...)** method.

Step B : When the time comes to create a new object then a user passes an reference onto the “factory” class along with the validity interval of the new condition object to the API.

Step C : The API makes all necessary preparation (determines where a new object should be created, etc.) and calls the user-supplied **doCreate(...)** method with the value of the clustering hint.

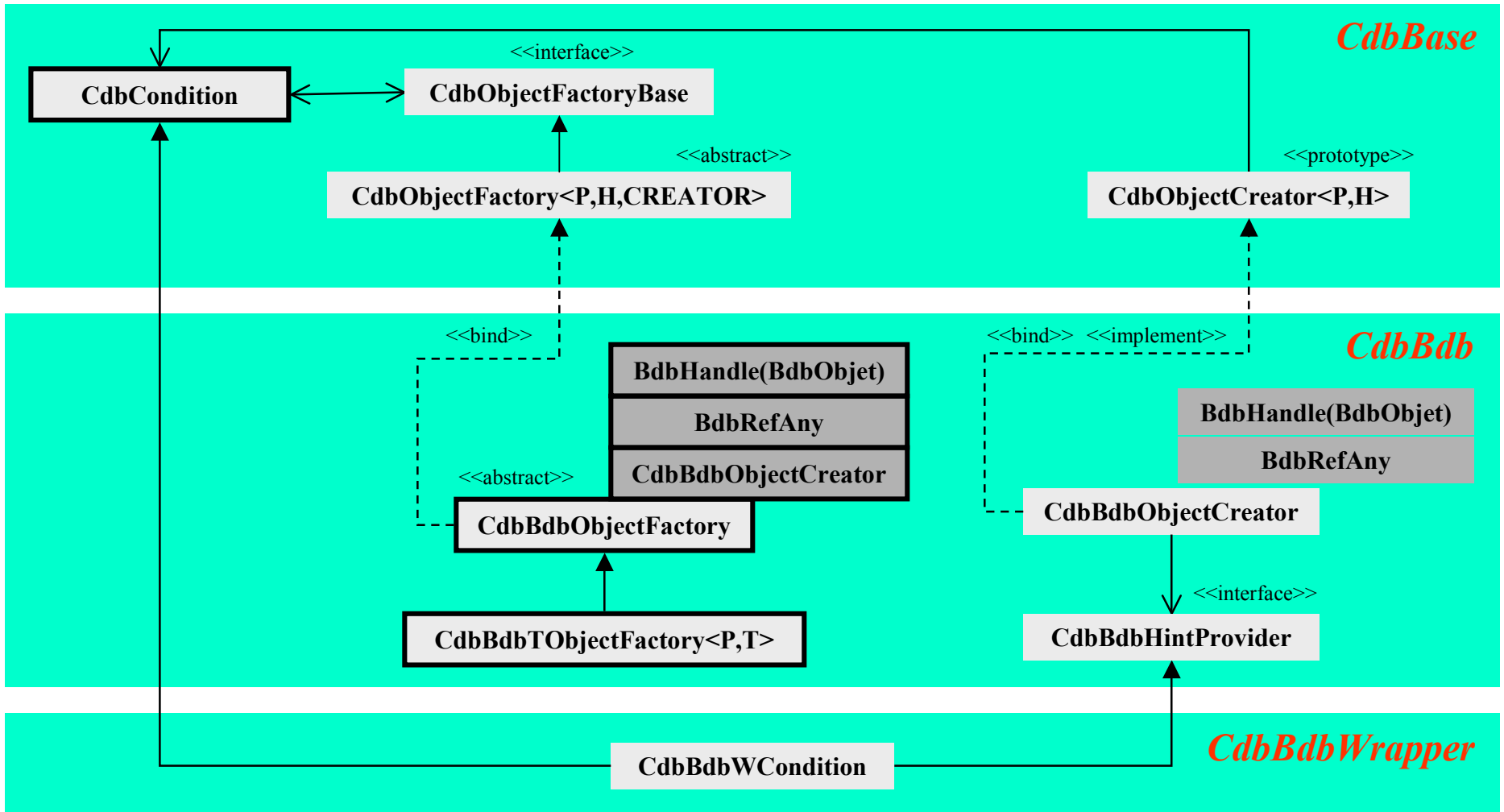
API : Data Clustering and Placement (2)



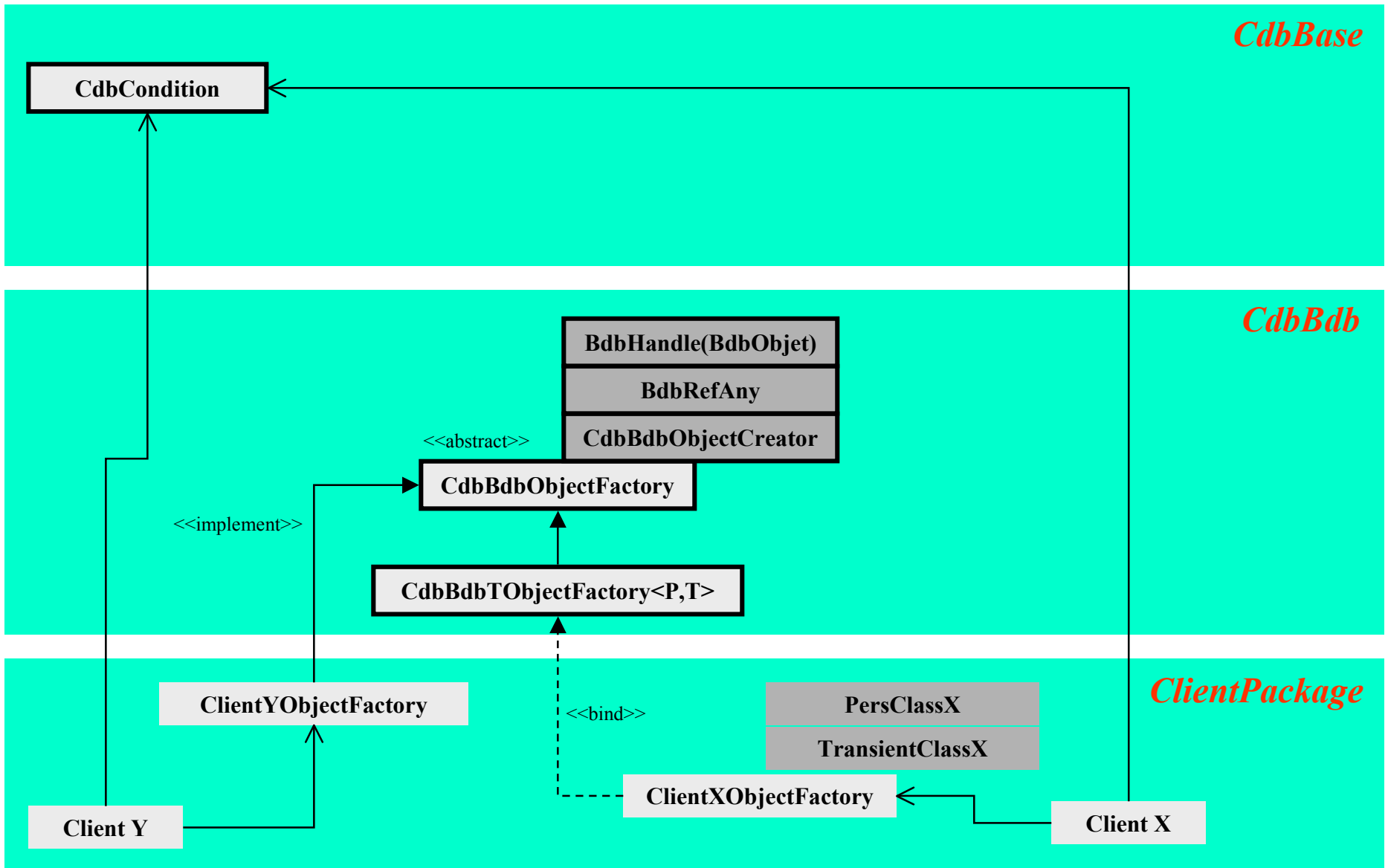
```
Status
Factory::doCreate( const Hint& h,
                   Object&    result )
{
    result = new( h ) MyClass( );
    return Status::Success;
}
```

API

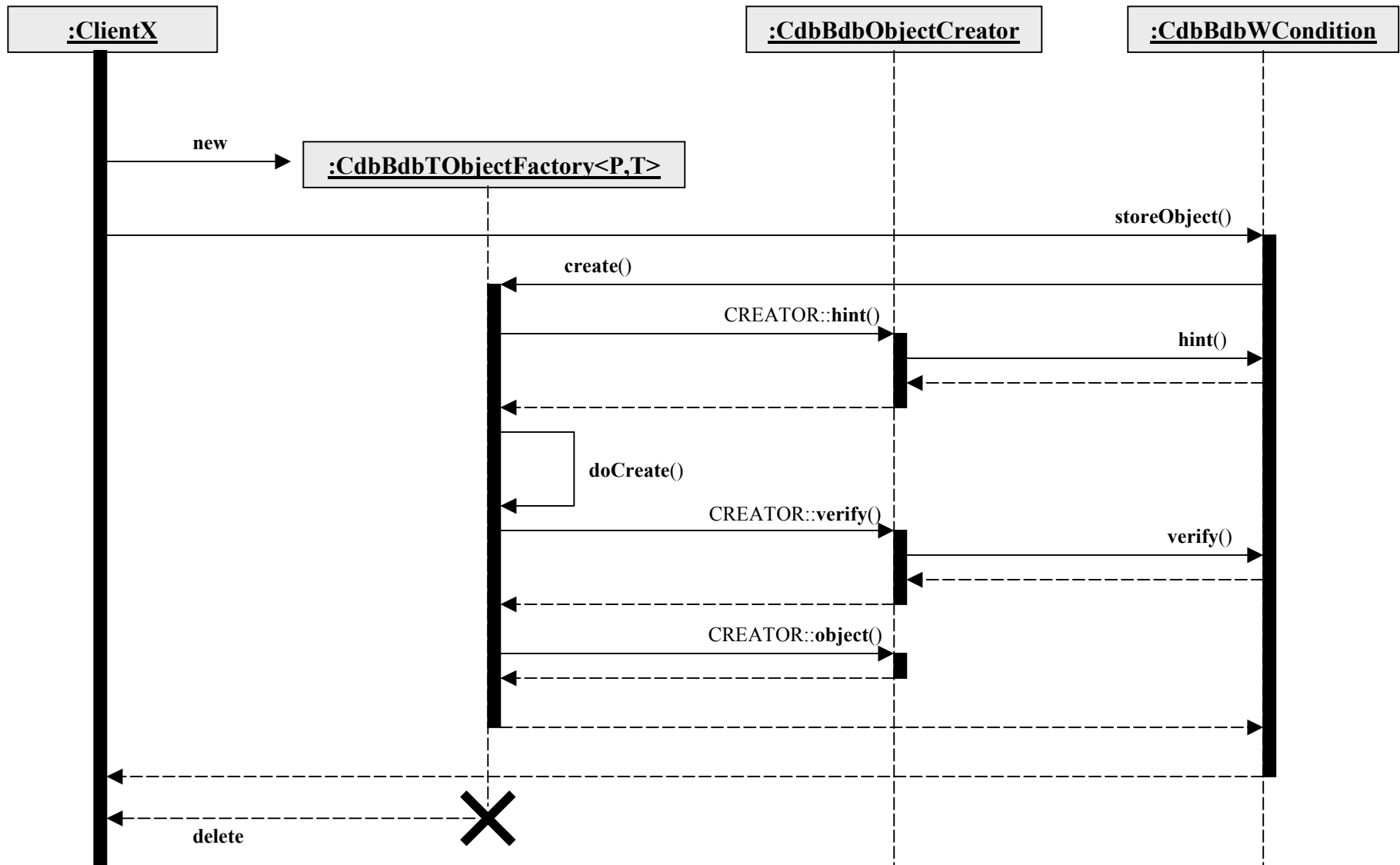
API : Storing Objects : Internal Architecture : Class Diagram



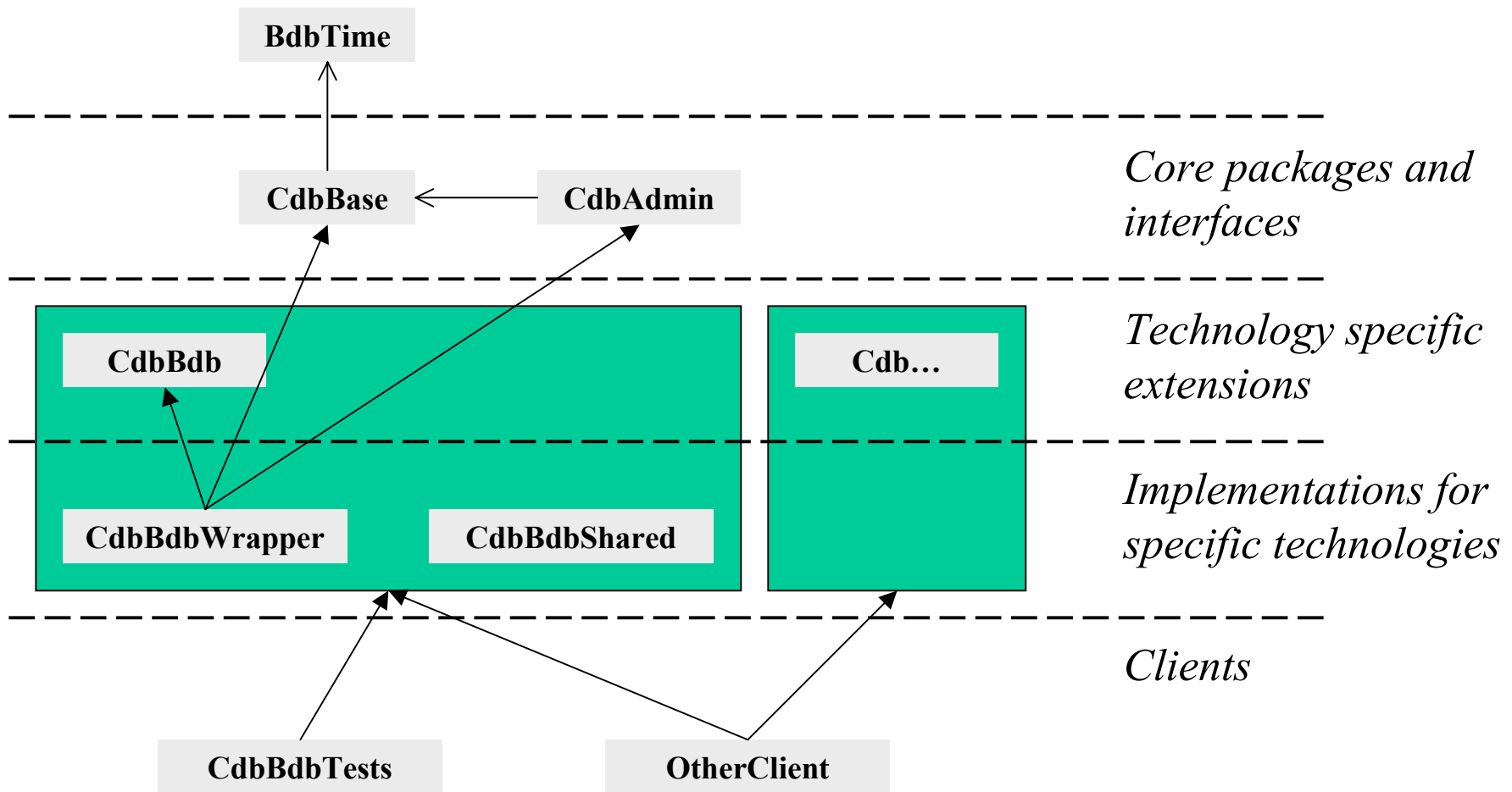
API : Storing Objects : Clients : Class Diagram



API : Storing Objects : Internal Architecture : Sequence Diagram



API : Packaging



API : Example : Finding a Condition Object

```
#include "CdbBase / CdbCondition.hh"
#include "CdbBase / CdbObject.hh"
#include "CdbBdb / CdbBdbObjectConvertor.hh"

#include "BdbTime / BdbTime.hh"
#include "BdbCond / BdbObject.hh"

// Step A : Locate a condition by its name

CdbConditionPtr cPtr;
if( CdbStatus::Success != CdbCondition::instance( cPtr,
                                                "/emc/EmcFooClassP" )) ...

// Step A : Locate a condition object for specified validity time (current
//           time in this example).

BdbTime currentTime;

CdbObjectPtr oPtr;
if( CdbStatus::Success != cPtr->findObject( oPtr,
                                           currentTime )) ...

// Step C : Convert a generic condition object into a persistent handle

BdbHandle(BdbObject) objectH;
if( CdbStatus::Success != CdbBdbObjectConvertor::narrow( objectH,
                                                         oPtr )) ...

if( BdbIsNull(objectH)) ...
```

API : Example : Storing a new Condition Object

```
#include "CdbBase / CdbCondition.hh"
#include "CdbBase / CdbObject.hh"
#include "CdbBdb / CdbBdbTObjectFactory.hh"

#include "BdbTime / BdbTime.hh"
#include "BdbCond / BdbDataListsP.hh"

// Step A : Locate a condition by its name

CdbConditionPtr cPtr;
if( CdbStatus::Success != CdbCondition::instance( cPtr,
                                                "/emc/EmcFooClassP" )) ...

// Step B : Establish specialized object factory

CdbBdbTObjectFactory< const char* const, BdbDataListsP >
    oFactory( "Hello Persistent World!" );

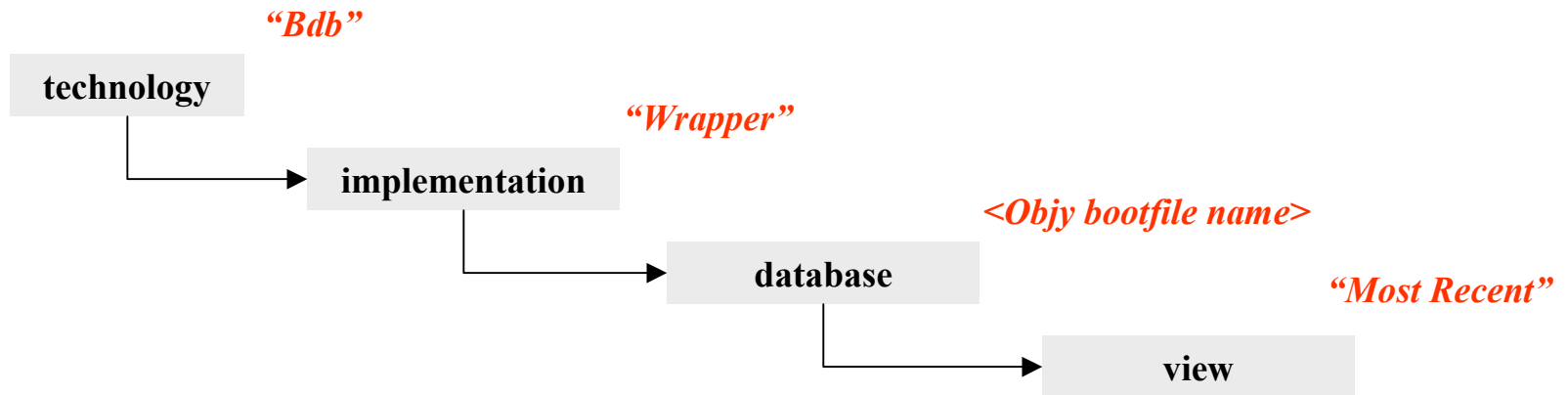
// Step C : Create and store a new object (the store method will tell the
//          factory to create a persistent object at suggested location.

BdbTime currentTime;

CdbObjectPtr oPtr;
if( CdbStatus::Success != cPtr->storeObject( oFactory,
                                             currentTime,
                                             BdbTime::plusInfinity )) ...
```

API : Job Configuration

An access path to a specific condition object is laying through the following choices to be made when dealing with the API:



In most cases (except very special ones) this process (from the perspective of clients' code) can be and must be simplified by introducing default values for the mentioned above parameters.

These defaults can be handled through the Job Configuration facility. The configuration can be specified either through a special environment variable or be loaded explicitly by the job at the beginning of its execution.

Persistent Data Structures

Technology: “*Bdb*”
Implementation: “*Shared*”

Persistent Data Structures : Kinds of Databases

Explicit support for 3 kinds of databases:

- **Master** (1)
- **Slave** (0..n)
- **Replica** (0..m)

“Master” knows about its “Slaves” and vs.

“Replica” is a full copy of the “Master”

“Master” and ‘Slaves’ each live in their own range of DBID-s much like we have now for IR2, OPR and REP-s. This allows to avoid conflicts and makes Data Distribution tools easier to implement.

Examples:

OPR : “Master”

IR2 : “Slave #1” of OPR

REP : “Slave #2” of OPR

Persistent Data Structures : Database Files (1)

The “bootstrap” mechanism for clients is based on the system (in terms of Objectivity) name of the following database files:

cdb_database_<origin>

A client knows its local DBID range (“origin”). Therefore it can get to the right database file.

*The contents of these files differ for “**Master**”, “**Slave**” and “**Replica**” (see Workbook for details). Here is a list of what’s in:*

Master : database identity, *list of “Slaves”, condition types definitions, list of conditions*, list of (local) views, “History” records, Data Distribution records.

Slave : database identity, back reference to “Master”, list of (local) views, “History” records, Data Distribution records.

Persistent Data Structures : Database Files (2)

The other database files are meant as a data storage for condition objects and the corresponding meta-data (intervals, revisions, etc.). These databases have the following names:

cdb_p_<id>

cdb_p_<id>_b_<block>

...

Data Clustering and naming the databases is done by the Condition/DB itself.

Database Placement is done by the ***Clustering Hint***.

Persistent Data Structures : Data Clustering Strategies

Principle 1 : Meta-data for a condition (partition) are stored in a single container.

Principle 2 : Condition objects for a specific condition (partition) are stored in a separate set of containers (no containers sharing as in the current Condition/DB).

Principle 3 : Meta-data container and its condition object containers are kept in the same database file.

Principle 4 : There is a variety of clustering methods for conditions (partitions) controlled by the Database Administrator.

Ensuring Self-consistency of Persistent Data Structures

Main idea:

There is an idea to implement persistent support to maintain the self-consistency of persistent data structures of the Cond/DB.

Each persistent component (including "condition type", "condition", "partition", "view", etc.) will have a "unique timestamp" (a value of BdbTime class, including nanoseconds) recorded persistently. Therefore, if someone brings a persistent component meant to be either a new one or an updated one then the consistency checks could be done to ensure that this component is the one expected and that it does not collide with already existing components.

One example of this would be the incremental snapshots between Italy (REPRO) and SLAC (where the "master" Condition/DB is located). The current logic requires:

- partitions to be created/planned at the "master" database only. Then they (partitions) are shipped to Italy for writing new RC conditions into it. If something goes wrong in Italy (data corruption, management problem, etc.) then their output (their partitions filled with RC-s) will be rejected by the 'master' at the attachment phase.

Another example - exchange conditions across databases. The proposed protection/identification logic will protect against attaching two different versions of the same condition.

Master-Slave Separation of Databases

Main idea:

To support the distributed and replicated Condition/DB the concept of the "master" and "slave" has to be introduced. Each of the databases involved in this process should know who is who.

The main reason to introduce this concept is to control the responsibility and access rights among a master and its slaves. Some operations may only be available to the master database. Another benefit would be to prevent any distributions across incompatible "master-master", "slave-slave".

Normally, when the database is created from the scratch it becomes the "master" one. Creating the "slave" federation is a special management procedure (a part of the data distribution tools).

However it would be better to consider a database initialization procedure supporting both "master" & "replica" modes. This can make data distribution a bit easier. See how it can work:

- when the database is created it gets the unique id (timestamp as a value of BdbTime).
- when the "slave" gets created it also gets a unique id of its "master" database.
- this would allow to control the correctness of the data distribution across master and its slaves.

IMPORTANT NOTE:

- An implementation of this (master-slave) concept has to be coordinated with the "DBID range".

Implementing partitions in Master-Slave databases

The main question to be answered is - how to design the persistent data structures of the new implementation to satisfy the distributed nature of the Condition/DB?

An answer to this question will affect the ways the synchronization (data distribution) schemes of the components of a single database distributed across multiple locations (Master-Slave) will be implemented. The general requirements to this synchronization are:

- It must not be too complicated
- Only the minimum set of components has to be involved
- It must be fast
- It must be robust

Another related topic is how the distributed database will be managed. Certainly some of the operations will only be possible in the "Master" database. Then they (modifications) will be propagated to slaves.

The "Database ID Ranges" should be used to implement Master-Slave separation.

Speaking about the design of data structures, it's worth to mention that some of these data structures must be replicated across the parts of the databases. The unfortunate consequence of this is:

- Some extra care will be needed to ensure the correctness of the replicated information in a course of the mentioned above synchronization procedures.

The first step on the way of the design is to identify what kind of information about partition is needed:

First of all we need the "Layout of Partitions", which is meant to describe what fraction of the two-dimension space (of the insertion and validity dimensions) is covered by each of known partition.

Then we need detailed information about each partition, including:

- Its limits in the insertion dimension
- Its limits in the validity dimension
- Its current status, which can be: open for update, closed/frozen, etc.
- Its owner, which in terms of the current implementation can be the "origin" (synonym "Database ID Range")
- Last time it (partition) was modified. This feature is needed to support the implementation and retrieval of values for the "State Identifier".

Databases (1)

`cdb_database_<origin>`

Each Condition/DB (both "master" and its "slaves") have this database file. This file plays the role of the "bootstrap" handle when the Condition/DB API opens the database. The minimum amount of information in these databases includes:

- If the current database is "master" or "slave"
- And if it's "slave" then who is the "master" of this slave

This database file not only provides the bootstrap information but also, depending on the kind of the database (master or slave) the specific information related to it.

How this file is used:

- Normally when an application needs to open the Condition/DB database it (application) will obtain its `<origin>` name from the "Database ID Allocation" facility.
- Based on that name the right instance of the `cdb_database_<origin>` database is open. The contents of this database varies depending on whether the local Condition/DB is the "master" or a "slave" one.
- This database will provide access to the rest of the data.

The current design assumes:

Each part ("master" or "slave") of the database will contain at least one (its own) database file with such name. In addition, every slave will have the master's file. The reason of this is that only the "master" database file provides the whole picture of how the Condition/DB is structured, and from which components it's made of.

For example, if our Condition/DB is distributed between OPR and REPRO, and OPR is the "master" one, then the following database files will coexist in the REPRO:

```
cdb_database_opr  
cdb_database_repro
```

Remember that we're talking about a physically distributed Condition/DB database, not a shared one. A shared database has just one `<origin>`. But the parts of it can be accessed simultaneously.

Databases (2)

`cdb_database_<origin> @ Master`

The "master" database is identified through the presence of a container called "Master". This container has the only object - the description of the "master", including the following:

- The creation time of the "Master", which also plays the role of the unique identification of this database.

There is a container called "History", which keeps a registry (in the chronological order) of some significant database-scope operations performed in the "master" database. This may include modifications done to the persistent structures (creating new condition types, conditions, partitions, views) as well as the information about the imports from "slave"-s with detailed description of these imports.

There is a container called "Types". Currently the only object in this container is called "Conditions", which is a registry of the known condition types and their characteristics.

There is a container called "Conditions". This is a registry of all the known conditions in the database. Each entry in this list is identified by name, which is derived from the creation time of the condition. For example: 3123456789_0123456789. Each entry also has built-in information about the creation time of the condition (also can be used as unique identifier), its type (see above), its description, etc.

There is a container called "Views", which is a catalogue of all known to the "Master" views.

NOTE: That according to the current strategy, the views can be also created in "Slave" databases. That implies that these views have to be registered in the "Master" database as a part of the "importation" procedure.

There is a container called "Partitions" with the layout of the partitions. This is the only place where this kind of information is available. There might also be an index for quicker access to desired partition. Each partition record has the following static information:

- Its identifier (a plain 16-bit number starting from 0)
- Its "size" in the 2-d space (insertion & validity)
- Its status (open/closed)
- Its creation time (can also be used for the consistency checking)
- Its owner (the name of the <origin>)

The dynamic (the one that gets updated) information about each partition is stored in a dedicated database file belonging to the owner origin of the partition. This will be described in more details in further slides.

Databases (3)

`cdb_database_<origin> @ Slave`

The "slave" database is identified through the presence of a container called "Slave". This container has the only object - the description of the "slave", including the following:

- The creation time of the "Slave", which also plays the role of the unique identification of this database.
- The identifier of the "master", including the <origin> of the "master" and its creation time (again for the purpose of the consistency checking).

As in case of the "master" database, the "slave" one also has a container called "History", which keeps a registry (in the chronological order) of some significant database-scope operations performed in the "slave" database. For example - when the "updates" from the "master" database were done, and what kind of updates they were.

Optionally there might be a container called "Views", which is a catalogue of locally (at the "Slave") views. This feature may not be available in the first implementation.

Databases (4)

The “System” container of databases:

Each database file should have the following container:

```
"System"
```

It will contain as minimum the following information:

- Condition/DB creation time, which also servers as its unique identifier.
- The database file descriptor, including the creation time of this particular database file, its type (bootstrap, partition, etc.)

It will also contain specific records depending on a database file type.

Mapping Partitions onto Objectivity Db Files (1)

General requirements:

Since partitioned conditions are most likely to be the major contributor into the overall size of the Condition/DB then their design must be flexible enough:

- to have a compact (in terms of the number of database files) size of the Condition/DB whenever it's possible. Ideally we can think about just one database file per partition.
- to be easily extendable when it's needed. This may happen if the amount of data related to a specific partition will exceed the reasonable capacity of a database file. Another case would be when the database file size will affect the recording/fetching performance of the database.

To address these requirements the design of the persistent structures should have "an open path" for the further extensions. At the very beginning we can start with a simplest solution with possible improvements if they will be required.

The following pages study various design options.

In practice all these options can co-exist in a single Condition/DB. Depending on the expectation and experience different partitions can be created with different placement policies.

Mapping Partitions onto Objectivity Db Files (2)

The “all-in-one” approach:

There is just one database per partition:

```
cdb_p_<id>
```

The database has the following containers:

```
"System"           - the description and status information about partition
"History"          - the history record related to that partition
"<cond>.MetaData"  - intervals, revisions, indexes, history, etc.
"<cond>.Objects"   - the actual condition objects
```

Where:

```
<id>      - is a partition identifier ("0", "1")
<cond>    - is a condition identifier (timestamp "3123123123.0456456456" or just number)
```

What's in the "System" container:

- The creation time of the partition. It's used for the self-consistency cross-checks.
- The timestamp when the partition was most recently modified.

COMMENTS: The basic idea of this approach is that partitioned conditions are always handled together as a whole management unit. When the partition database will get too big then the corresponding condition can be closed and a new one will be open.

The benefits of this approach are not quite obvious!

A variation of the “all-in-one” approach:

We can think about storing the condition objects themselves at a separate database file. Therefore we may have the following databases:

```
cdb_p_<id>
cdb_p_<id>_objects
```

The "System" container of the "cdb_p_<id>" database should have the hint indicating this option. The other containers will be the same.

Mapping Partitions onto Objectivity Db Files (3)

The “multi-block-all-in-one” approach:

This is another variation of the “all-in-one” approach.

If putting all conditions into a single database file makes the files too big in size then slightly different approach can be applied. The idea is to distribute conditions across a number of database file, or “blocks”:

```
cdb_p_<id>  
cdb_p_<id>_b_<block>
```

The first database will only contain the “System” and the “History” containers. The others will have “System” and the corresponding condition specific containers:

```
"<cond>.MetaData"    - intervals, revisions, indexes, history, etc.  
"<cond>.Objects"     - the actual condition objects
```

The “System” container is needed for the integrity checking only. It will also identify the contents of the database.

The number of conditions per block (or total number of blocks in a partition) is a configuration number defined at a partition creation time. This number is recorded somewhere at the partition description block (the “Partition Layout” is the best candidate).

In order to tell a transient code which block has which condition the “System” container of the “cdb_p_<id>” should provide a mapping table.

Clients:

Natural clients of this kind of mapping are those conditions, which are supposed to be exchanged all together in a whole block. One example would be conditions used by analysis jobs.

Mapping Partitions onto Objectivity Db Files (4)

The “incremented-all-in-one” approach:

This is one more way to improve the “all-in-one” approach.

In some case it may be needed to split a partition onto the “increments” in the insertion time dimension, so that smaller “increments” of the partition could be exchanged between the “master” and “slave” databases. To implement this idea a single partition database file can be split onto a sequence of “increments”:

```
cdb_p_<id>  
cdb_p_<id>_i_<increment>
```

The first database will only contain the “System” and the “History” containers. The others will have “System” and the corresponding condition specific containers for each increment:

```
"<cond>.MetaData"    - intervals, revisions, indexes, history, etc.  
"<cond>.Objects"     - the actual condition objects
```

The “System” container of the “cdb_p_<id>” database will keep track for all the increments. So that the transient code will be able to find out a required condition object at the partition.

Both metadata (including history, revisions, etc.) and condition objects of an increment represent self-consistent set of information.

Clients:

This kind of mapping will be useful for the rolling calibration conditions (partitioned in the validity dimension to be updated simultaneously at a “master” and “slave” databases). This is especially useful for the fast growing conditions.

The main difference of “increments” from “partitions” is that increments can be generated by a “slave” database.

Mapping Partitions onto Objectivity Db Files (5)

The “one-partition-one-condition-one-file” approach:

The basic idea is that each condition is represented in a partition by a separate database file. Therefore we have the following database files:

```
cdb_p_<id>  
cdb_p_<id>_c_<cond>
```

The first database will only contain the “System” and the “History” containers. The others will have “System” and two condition specific containers:

```
"<cond>.MetaData"    - intervals, revisions, indexes, history, etc.  
"<cond>.Objects"     - the actual condition objects
```

The “System” container is needed for the integrity checking only. It will also identify the contents of the database.

Clients:

Slowly changing conditions whose are supposed to be shipped on their own as a subset of the whole Condition/DB.

The further granularity for this kind of conditions using the previously described “incremental” approach does not seem to be useful due to small amount of data in these files. Although we do not rule out this option.

Mapping Partitions onto Objectivity Db Files : Summary

The Summary:

All the previously discussed options have a number of common features:

- They apply only to those conditions that (according to their type) can be partitioned. The other conditions are mapped onto the database using different techniques. This will be discussed later.
- All conditions of this group are partitioned together.
- Their database names begin with: "cdb_p_<id>".
- They all have at least one database "cdb_p_<id>" file with at least two containers "System" and "History".
- They assume that the data mining code will be smart enough to deal with any of these implementations.
- A client is given a choice (and instrumentation) to create partitions of his choice.

Specifically the following mapping options have been discussed:

```
all-in-one  
separated-objects-all-in-one           <- practically useless  
multi-block-all-in-one  
incremented-all-in-one  
one-condition-one-partition-one-file
```

Further variations/combinations of these approaches are also possible. For example:

```
incremented-multi-block-all-in-one       // cdb_p_<id>_b_<block>_i_<increment>  
incremented-one-condition-one-partition-one-file // cdb_p_<id>_c_<cond>_i_<increment>
```

Mapping non-partitioned conditions onto Objectivity Db Files (1)

General Requirements:

The design is expected to be flexible enough to provide the following options in the mapping:

- a) To map a condition onto a single database file
- b) To have multiple "increments" of the condition's database file
- c) To map multiple conditions (a block of conditions) onto a single database file
- d) To have multiple "increments" for the block of conditions.

These requirements lead to the following mapping strategies:

one-condition-one-file
incremented-one-condition-one-file
multi-block
incremented-multi-block

The specifics (database naming, etc.) for each of the mentioned above mapping strategies are being discussed on the subsequent pages.

In general it's up to the Condition/DB manager how to split the non-partition conditions. The corresponding management API and utilities will provide the required support.

At the implementation level the information about the mapping policy and the corresponding parameters of this policy (number of blocks or number of conditions per block, etc.) is maintained at the following "master" database file:

`cdb_database_<origin>`

Specifically this information can be kept either at "Conditions" container of this database or some other container solely dedicated to the clustering.

IMPORTANT NOTE: The current design assumes that non-partitioned conditions can only be created and modified in the "master" database. This requirement can be reconsidered later as more experiments and related discussions will be conducted.

Mapping non-partitioned conditions onto Objectivity Db Files (2)

The “one-condition-one-file” approach:

The basic idea is that each condition is represented has its own separate database file:
Therefore we have the following database files:

```
cdb_c_<cond>
```

The database will only contain the following containers:

```
"System"
"<cond>.MetaData"    - intervals, revisions, indexes, history, etc.
"<cond>.Objects"     - the actual condition objects
```

The “System” container is needed for the integrity checking only. It will also identify the contents of the database.

Clients:

Slowly changing conditions whose are supposed to be shipped on their own as a subset of the whole Condition/DB.

Mapping non-partitioned conditions onto Objectivity Db Files (3)

The “incremented-one-condition-one-file” approach:

This is an extension of the previously described “one-condition-one-file” approach. The only difference is that the whole condition is spread between so called “increments”. Each of the increments includes a certain range of “revisions”. Here is how this idea maps onto database file names:

```
cdb_c_<cond>_i_<increment>
```

There must be at least one database of this kind (with increment number 0):

```
cdb_c_<cond>_i_0
```

NOTE: In theory we can think about slightly different names:

```
cdb_c<cond>_i<increment>
```

or may be:

```
cdb_c<cond>i<increment>
```

Clients:

Quickly changing conditions whose are supposed to be shipped on their own as a subset of the whole Condition/DB. One good example of these conditions would be “online calibrations” produced by IR2.

For the sake of simplicity (and flexibility in case if we decide to go “incremented” from the just “one-condition-one-file” one) it would be wise to have just a single pattern in database names - the one used by incremented approach.

So there will be just one database file per condition - its the only increment.

Mapping non-partitioned conditions onto Objectivity Db Files (4)

The “multi-block” approach:

According to this approach many conditions can be placed into a single database file, so that the overall number of the database file in the database will be less than in case of “one-condition-one-database” mapping strategy. The resulting databases are expected to hve the following names:

```
cdb_b_<block>
```

There must be at least one database of this kind (with block number 0):

```
cdb_b_0
```

Each file will have the following containers:

```
"System"  
"<cond>.MetaData"    - intervals, revisions, indexes, history, etc.  
"<cond>.Objects"     - the actual condition objects
```

If a condition is put into a block then the corresponding placement record in the “Containers” container of the “master” database will have a number of the block.

Clients:

Slowly changing conditions whose are not supposed to be shipped on their own.

Mapping non-partitioned conditions onto Objectivity Db Files (5)

The “incremented-multi-block” approach:

Similarly to the “incremented-one-condition-one-file” approach this one can be used to break the information about a block of conditions onto a set of “increments”. The corresponding database names will look like:


```
cdb_b_<block>_i_<increment>
```

If a condition is put into a block then the corresponding placement record in the “Containers” container of the “master” database will have a number of the block.

The “System” container of the “0” block will have the total number of increments in the block. We can possibly put some extra information regarding each of these increments into this container. This may include the “insertion” time range for each of the increments, etc.

Clients:

Relatively quickly changing conditions whose are not supposed to be shipped on their own.



In fact this technology makes a pure “multi-block” one just a subset of the “incremented” one.

Therefore for the sake of simplicity (and flexibility in case if we decide to go “incremented” from the just “multi-block” one) it would be wise to have just a single pattern in database names - the one used by incremented approach.

So there will be just one database file per block - its the only increment.

Non-partitioned conditions in Master/Slave databases

What's this question about?

At some point of time it might be needed to delegate the update of whole (not just partitions) non-partition conditions from the "master" to one of its "slaves". In fact this is exactly what we have now for the IR2, OPR and REP[x] federations.

There is no doubt that this kind of functionality will be asked by clients (even if they don't even know about this now). One way to resolve this issue in the new design would be to follow the same approach as it was suggested for "partitions", namely to specify the "origin" for each of the conditions found at the container named "Conditions" in the "master"'s database file:

```
cdb_database_<origin>
```

The logic here is very simple. When a "slave" database gets initialized then the "initialize" utility will browse through the list of conditions and will create the corresponding database file in the local Database ID range of the "slave" database.

The Replica kind of databases

The Replica database is:

In the reality of the distributed database environment like we have at SLAC some databases are only used as copies of the actual ones where the data are generated. These copies are called replica databases.

A replica database is a read-only snapshot of some "master" or "slave" database. But unlike a trivial full snapshot the replica can be updated incrementally. In addition it can also keep a track of its increments (when was the last time the update was done, and what kind of information was updated).

First example of the replica kind of databases would be the Conditions/Db of the current "Physics Analysis Federation". This is going to be a replica of the "master" database of OPR.

Another example of a replica of the "slave" federation would be the "Ambient" federation. This federation is updated from the Condition/DB of the current IR2 federation on a regular basics (every run).

POTENTIAL PROBLEM: The proposed solution to use replica in Ambient has a potential conflict in case if this replica will also be updated from the "master" (which is going to be OPR).

Technically "replica" has the same very similar set of data structures of a "slave" one. The only exception is that there is no "Views" container in its main database file. This is because there is no data associated with replica in this database.

Catalogue of Persistent Entities (1)

DATABASE	CONTAINER	OBJECT	CLASS	COUNT	
con_database_<origin> @ Master	"System"	"Id"	CdbBdbSDatabaseIdP	1	
		"File"	CdbBdbSFileDescP	1	
		"Master"	CdbBdbSMasterDescP	1	
	"History"	"History"		CdbBdbSHistoryRegistryP	1
				CdbBdbSHistoryEventP	1..n
				<index for history events>	1
	"Types"	"ConditionTypes"		CdbBdbSConditionTypesRegistryP	1
				CdbBdbSConditionTypeP	0..n
	"Conditions"	"Conditions"		CdbBdbSConditionsRegistry	1
				CdbBdbSConditionP	0..n
	"Partitions"	"PartitionsLayout"		CdbBdbSPartitionsLayoutP	1
				CdbBdbSPartitionP	0..n
				<index for partitions>	1
	"Views"	"Views"		CdbBdbSViewsRegistryP	1
			CdbBdbSViewP	1..n	

cdb_database_<origin> @ Slave	"System"	"Id"	CdbBdbSDatabaseIdP	1	
		"File"	CdbBdbSFileDescP	1	
		"Slave"	CdbBdbSSlaveDescP	1	
	"History"	"History"		CdbBdbSHistoryRegistryP	1
				CdbBdbSHistoryEventP	1..n
				<index for history events>	1
	"Views"	"Views"		CdbBdbSViewsRegistryP	1
				CdbBdbSViewP	1..n

Catalogue of Persistent Entities (2)

DATABASE	CONTAINER	OBJECT	CLASS	COUNT	
con_database_<origin> @ Replica	"System"	"Id"	CdbBdbSDatabaseIdP	1	
		"File"	CdbBdbSFileDescP	1	
		"Replica"	CdbBdbSReplicaDescP	1	
	"History"	"History"		CdbBdbSHistoryRegistryP	1
				CdbBdbSHistoryEventP	1..n
				<index for history events>	1
con_b_<block>	"System"	"Id"	CdbBdbSDatabaseIdP	1	
		"File"	CdbBdbSFileDescP	1	
	"<cond>.MetaData"	"OriginalIntervals"		<a list of original intervals>	1
				CdbBdbSOriginalIntervalP	0..n
				<an index of these intervals>	1
	"Revisions"	"Revisions"		CdbBdbSRevisionsRegistryP	1
				CdbBdbSRevisionP	1..n
				CdbBdbSVisibleIntervalP	0..n
	"History"	"History"		CdbBdbSHistoryRegistryP	1
				CdbBdbSHistoryEventP	1..n
			<index for history events>	1	
"<cond>.Objects"	n/a		BdbObject	0..n	

Catalogue of Use Cases : Creation/Initialization

Explicit Creation/Initialization procedures:

This class of procedure will create the corresponding entities in the database upon a user's request. So we can treat these operations as the database management ones. Each of these operations requires certain persistent entities to exist in the database.

ID	ENTITY	REQUIRES	COMMENTS
---	-----	-----	-----
1	"master" database		This is the very first operation to be done. None of "cdb_*" database should be present in the federation. This will also create the default view.
2	"slave" database"	1	The federation must be loaded with a total snapshot of the corresponding "master" database.
3	"replica" database	1	The federation must be loaded with a total snapshot of the corresponding "master" or "slave" database.
4	condition type	1	
5	partition	1	This step is only required for partitioned conditions.
6	view	{1 2}	It's not needed if the only view is the default one. Remember, that views can also be created in "slave" federations.
7	condition	4(,5)(,6)	Partition is only required for partitioned condition types. The view is only required if it's not going to be the default one.
8	revision	7	

These procedures is described in details on the next pages.

Containers: “<cont>.MetaData”

What’s in this container:

These containers keeps meta-data related to the validity of objects of the corresponding (container identifier in the name) condition (a partition).

Various options have been studied to answer the following questions:

- *how to represent intervals*
- *how to index these intervals for efficient access*

The final choice:

- *prefer embedded classes where it’s technically possible*
- *develop “**paged V-Array**” for efficient storing and retrieval of embedded objects*
- *develop “**optimized B-Tree**” with “**fuzzy**” logic for efficient indexing of intervals in 1-dimension space of either **insertion** or **validity** timelines.*

Containers: “<cont>.MetaData” : Data Structures (1)

NOTE: *The same structures are used for non-partitioned conditions (all in one containers) and partitioned ones.*

*There are just one named (in a scope of a container) **Registry** object providing location to other data structures of the container.*

The other data structures include:

Original intervals:

- *describe the **original** client's intention for stored objects (object reference and its validity interval)*
- *stored as a (**paged V-**) array*
- *indexed through (**optimized B-tree**) index in the **insertion** time dimension*

*The index is used for partitioned conditions to find out an object matching specified value of a “**partition modification time**” (see **State Identifier**)*

Containers: “<cont>.MetaData” : Data Structures (2)

The other data structures include (continued):

Revisions:

- *stored as an (paged V-) array*
- *indexed through (optimized B-tree) index in the **insertion** time dimension*
- *there is at least one (topmost) revision in a container*

*Each revision represents a collection of **visible** intervals pointing back to original ones.*

*Collections are also indexed through (optimized B-tree) index in the **validity** time dimension.*

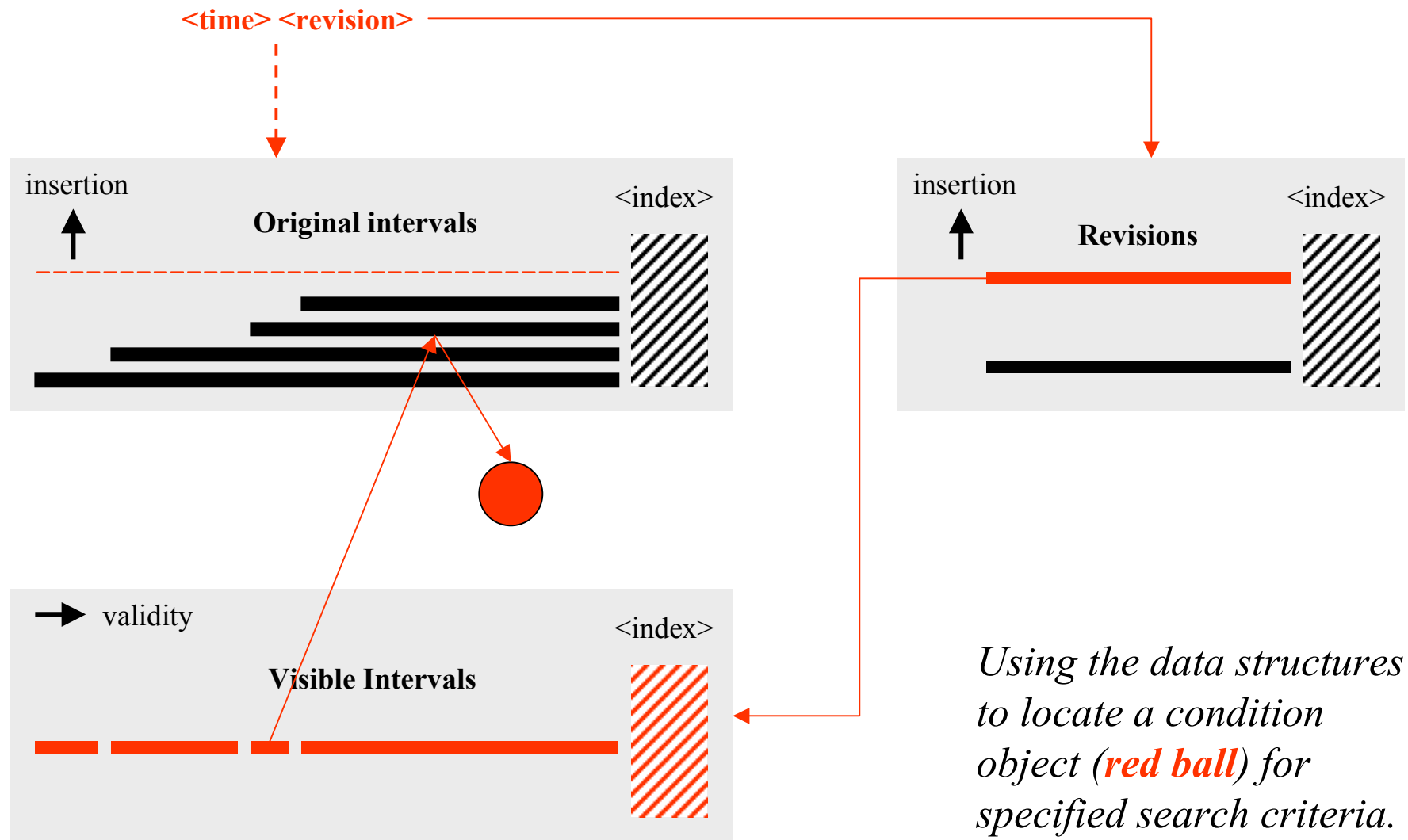
Containers: “<cont>.MetaData” : Data Structures (3)

The other data structures include (continued):

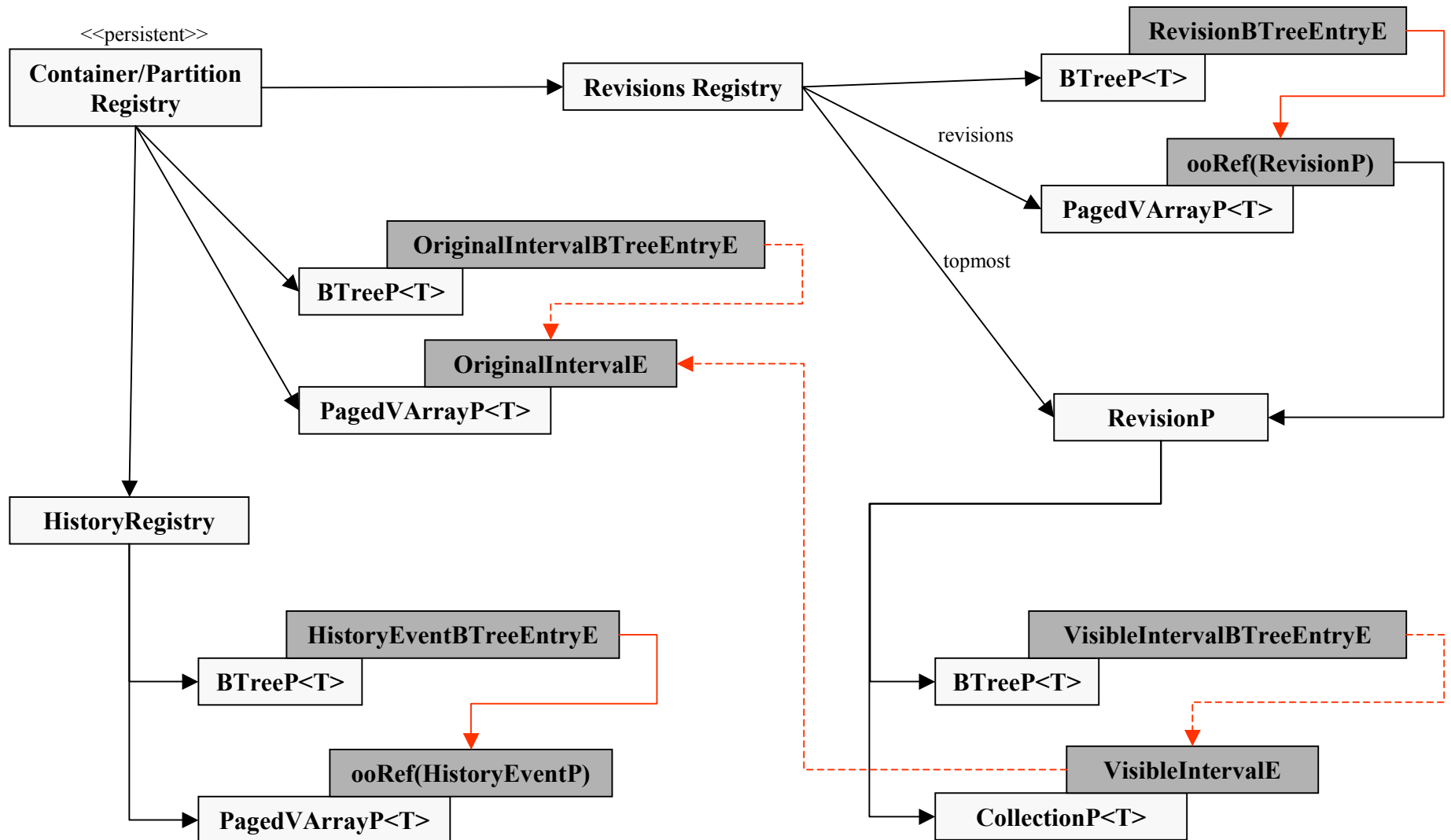
History:

- *describe various operations and their parameters modifying the contents of the container (storing new objects, creation of revisions, etc.).*
- *stored as an (**paged V-**) array*
- *indexed through (**optimized B-tree**) index in the **insertion** time dimension*
- *there is at least one (initial) record in this list*

Containers: “<cont>.MetaData” : Data Structures (4)



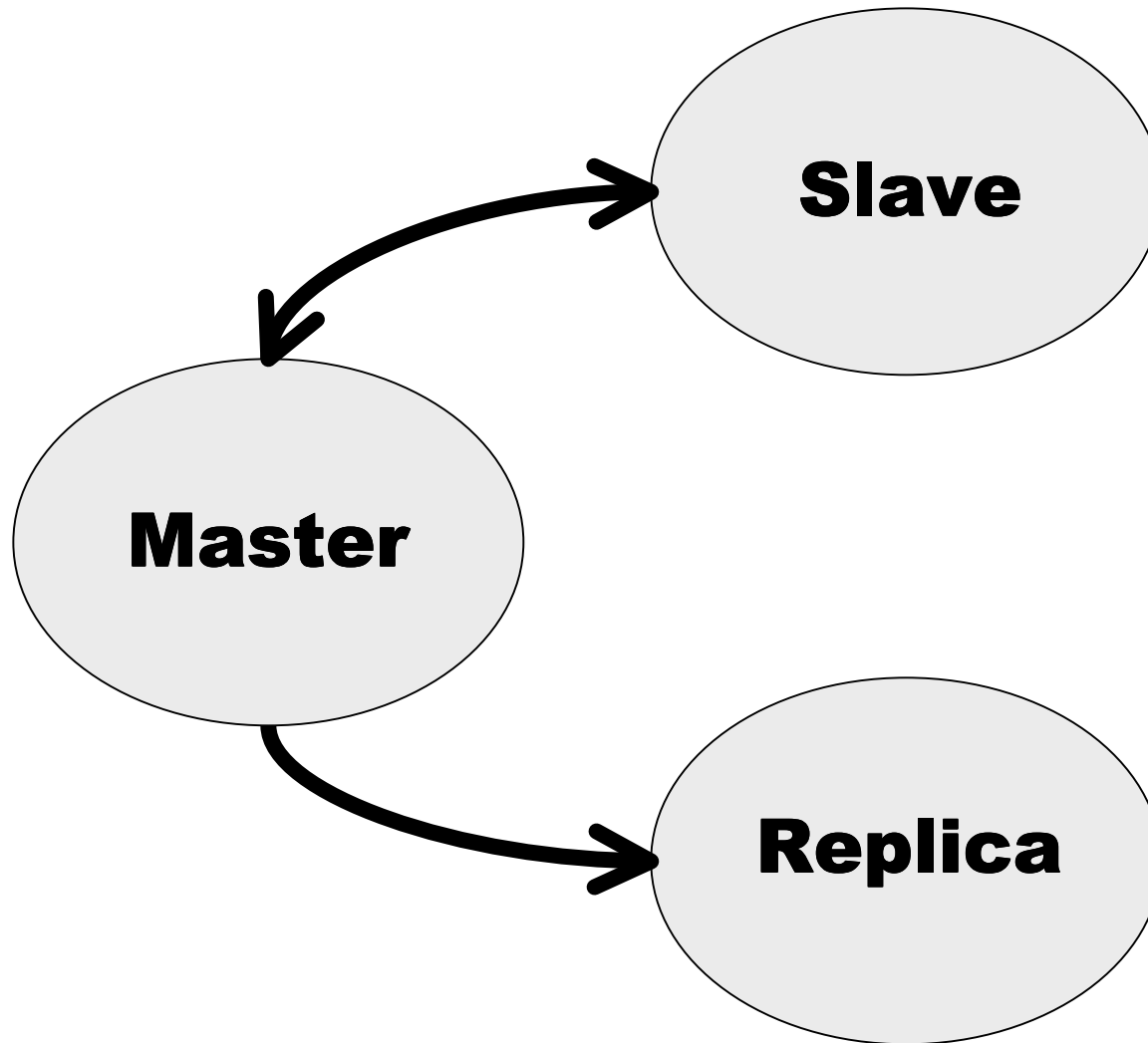
Containers: “<cont>.MetaData” : Class Diagram



Data Distribution Issues

Technology: “*Bdb*”
Implementation: “*Shared*”

Data Distribution : Kinds of Databases



Data Distribution : Most Optimal Approach

The idea:

Due to growing complexity of the data partitioning (as a sacrifice to the greater flexibility) in the new Condition/DB design the database management:

`"...must never be done by hands..."`

It means that all the import/export operations, including the data transfer itself, have to be done through special utilities in combinations with specially developed servers where it would seem appropriate.

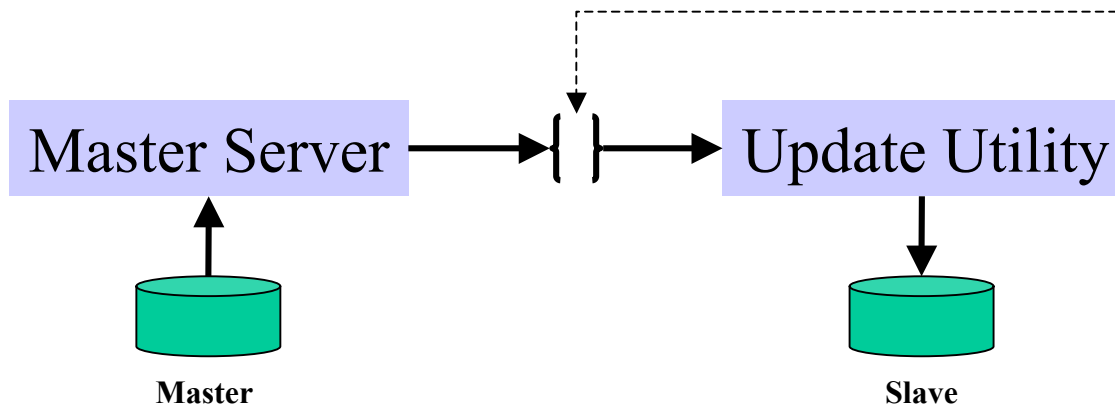
No more direct use of the "cp" to transfer database images!

The corresponding support for the (Condition/DB) database integrity has to be designed and implemented.

COMMENT: Ideally the clients/managers should not even worry about database files. The actual data placement will be hidden behind the API and the corresponding management utilities. The data partitioning strategies will play a role of technology-independent hints.

The server-utility protocol will transfer both information about the databases to be updated/added and the database images themselves (as if a plain copy is done).

The benefit is that these images will only be attached if nothing is going to be broke. This cure the famous problem of the "human error" when handling the database files.



Data Distribution : Realistic Approach (1)

Phase I : *Implement an **export** utility to look into a database (**Master, Slave or Replica**) and to get a list of database files matching specified exportation criteria.*

For example:

*“**database files updated since <data&time> of the previous export**”*

*The databases (**Master, Slave, Replica**) should “remember” the dates when they were updated and what kind of update (from which database and what exactly) it was in each case.*

*The resulting list of database files along with meta-information about their contents will be used to create a **TDF** dataset.*

*This dataset will be unpacked into a target database file system to replace old databases with their updated versions, and an **import** utility will be run against the target database to update local data structures.*

Data Distribution : Realistic Approach (2)

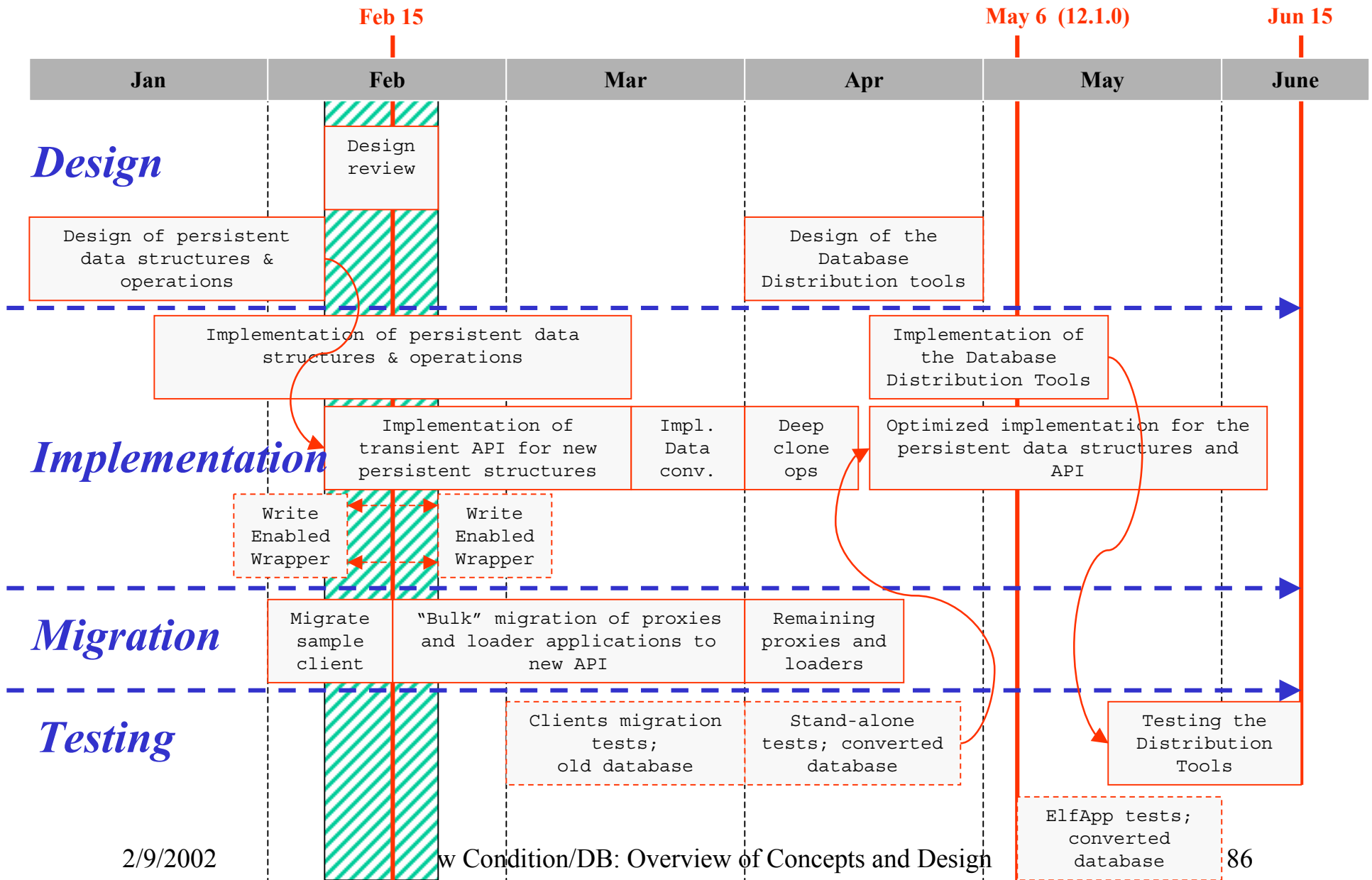
Phase II: *Introduce Data Distributed servers (CORBA, TCP, etc.) support for distributed databases to gather information about these databases from remote locations.*

This can be a helpful tool to “compare” databases’ contents and to determine which database files and what kind of information needs to be updated.

Phase III: *Implement even “smarter” version of the above mentioned server to perform the databases contents transfer between remote locations w/o involving traditional (TDG & snapshots based) Data Distribution Tools.*

Deployment Schedule, Documentation, Etc.

Implementation & Deployment Schedule (v1)



Implementation & Deployment Schedule (v1) : comments (1)

The notes on the meeting held on January 28:

1. Design review

Igor will arrange a 4 hrs meeting some day next week (to be discussed) here at LBL to review the design of persistent data structures of the new Condition/DB. We need this step to avoid potential hidden rocks at the implementation stage and/or with the scalability/performance of the new Condition/DB. In addition to the participants of the current mail thread it would be really useful to invite Dave Quarry (who also has expressed his interest to learn more about this project) and other experts (Steve Gowdy, Adil, Artem, someone from online databases (Andy Salnikov), who else?).

2. "Wrapper" implementation.

It has been realized that we do need WRITE-capable version of the "Wrapper" implementation for the "bulk" migration of proxies and loaders. The main reason of this is that developers of the proxies and loaders will need a fully functional (in terms of READING/WRITING) database in order to test their code. Another justification is that we can consider this "Wrapper" implementation as a "fallback" option for the production if the new one will be delayed beyond the acceptable (for the production) limits.

I (Igor) believe that this feature can be implemented and fully tested within a week. This is why I didn't pay too much attention to this feature in the past.

The right time to finish this work would be by March the 1-st (keeping in mind the "bulk" migration goal) or (as an option) the middle of February.

3. Most important dates in the schedule

Middle (or last week) of February:

We will identify classes/packages to be migrated. Then we will prepare a "migration pattern(s)" suitable for most of the clients. This will let us to use PC-s to offload Igor (...start beating the drum...). Special cases will be handled later (first weeks of April) at a separate stage. The whole clients migration process must end up by May 6 (release 12.1.0).

(to be continued on the next slide)

Implementation & Deployment Schedule (v1) : comments (2)

The notes on the meeting held on January 28:

3. Most important dates in the schedule (cont.)

May 6-th (release 12.1.0):

By this day we expect:

- a) Both the "wrapper" and the core "new" implementations be ready (for the release).
- b) The clients migration to be finished.
- c) The corresponding "deep clone" operations be implemented for 15 (or so) "composite objects" (the ones like the old "EmcCalibDict").
- d) The "old" -> "new" data conversion tools be ready.

What we do NOT expect:

- e) The Data Distribution tools
- f) The optimized implementation of the data structures and internal (data mining and storage) algorithms of the new Condition/DB
- g) Full-scale quality tests done (Elf in OPR-alike MP (Multi Processing, not Military Police :-)) environment).
- h) Final migration of existing namespace of conditions to the new (hierarchical) ones. For example:

```
"pep PepBeamsP"  
-> "/online/pep/PepBeamsP"  
  
"pep PepBeamSpotCal_Bhabha"  
-> "/RollingCalibrations/pep/PepBeamSpotCal_Bhabha"
```

Middle of June (15):

We expect 3 of 4 steps mentioned as (e), (f), and (h) above to be fully implemented.

The step (g) (full scale test in the Test-OPR federation against the fully converted database) can be done at this time.

This is a day when we make a final decision whether the new implementation (not just the wrapper one) goes into production on July 1.

(to be continued on the next slide)

Implementation & Deployment Schedule (v1) : comments (3)

The notes on the meeting held on January 28:

4. Concerns

The main one is what to do if the proposed schedule slips beyond acceptable (for production) limits. A possible "fallback" solution we have in mind is to keep using the "wrapper" implementation, which would not break anything (fully backward compatible with existing **BdbCond**). This will buy us some extra time (1 month) to find out an appropriate solution.

Dave Brown also has expressed concern about tight schedule in a period of April-May months. A possible solution would be to involve someone else to the implementation of the Data Distribution tools (see section "Manpower" for details).

5. Manpower

Some help in implementing the new Data Distribution tool would be very helpful to keep up with the schedule. Jacek proposes to discuss this with other members of BDB group at SLAC (Artem and Adil) at the next BDB meeting (presumably this Thursday).

Dave Brown and Igor will work together on the client's code (proxies and loaders) migration issue. As we get most common migration patterns then we may involve PC-s in the migration process.

The core design and implementation is left on Igor.

Where to get more information on this subject...

Documentation:

“Critical Analysis of the current Condition/DB in BaBar”

<http://www.slac.stanford.edu/~gapon/CriticalAnalysis.htm>

“Some ideas on the new Condition/DB design for BaBar (draft)”

<http://www.slac.stanford.edu/~gapon/proposal.htm>

DOXYGEN generated documentation + manuals (HTML)

<http://www.slac.stanford.edu/~gapon/projects/NewCondDb/>

Code (BaBar CVS packages):

- CdbBase** - *core classes and interfaces*
- CdbAdmin** - *administrative classes and interfaces*
- CdbBdb** - *specific API extension for “**BDB**” (Objectivity) technology*
- CdbBdbWrapper** - *“**Wrapper**” implementation of the “**BDB**” technology*
- CdbBdbTests** - *examples and verification tests for “**BDB**” technology*