

CDB – Distributed Conditions Database of the BaBar Experiment

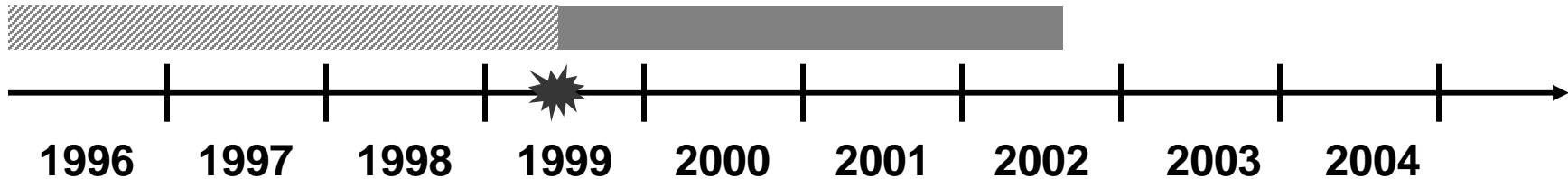
Igor A. Gaponenko (LBNL)

IAGaponenko@lbl.gov

Introduction

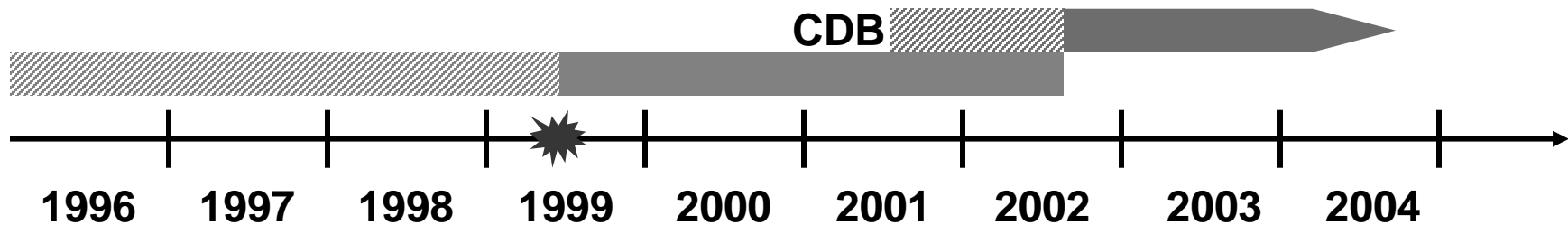
- **CDB is the second generation Conditions Database of the Experiment**
- **Its role is:**
 - To store time sensitive information about hardware and software environment (hence: *conditions*) in which detector *events* get acquired, modeled, processed and analyzed
 - Conditions and events are linked indirectly via two keys:
 - Primary: *event time (validity time in CDB)*
 - Secondary: *unique state identifier* of CDB or a *revision* of a particular condition
- **Same product (database and software) is used in:**
 - ONLINE : DAQ, Detector Control, Level-3 Trigger
 - OFFLINE : Reconstruction, Simulation and Analysis
- **Kinds of information stored:**
 - Detector alignments, constants, electronics wiring maps, calibrations, etc.
- **Information update frequency:**
 - Varies from minutes and hours (for calibrations) to months and years (alignments and constants)

An Original Conditions Database



- **Persistent technology: Objectivity/DB**
- **Went through a number of evolutionary improvements**
 - For functionality, scalability and performance
- **Areas of major problems (by 2001)**
 - General design
 - A conceptual model of the database was too simple to meet emerging requirements
 - Database was not specifically designed to be used in a distributed data processing chain
 - Its API
 - Was not persistent technology neutral
 - Was exposing an implementation of metadata
 - Performance and scalability
 - Was poor, in particular for automatically produced OFFLINE calibrations

CDB - The Second Generation Database



- **A product of a deep redesign**
 - “Critical mass” of dissatisfaction with the original database (by 2001)
 - The evolutionary improvements approach failed in addressing new challenges of the Experiment!
- **Based on an experience of the first 2 years of the Experiment**
- **Still uses Objectivity/DB (current implementation)**
 - Brand new metadata
 - Flexible data clustering
 - Inherited old user defined schema
 - Inherited previously stored condition objects (payload)
- **Opened a path for other persistent technologies be used in the database**

An Overview of CDB...

CDB : New Design

- **CDB is a distributed database**
 - From the ground-up, CDB is designed to serve applications in a distributed data processing chain of the Experiment
- **CDB is based on a revised logical model:**
 - A concept of the *origin*
 - A 2-D space (*validity* and *insertion* time) for time sensitive elements
 - A concept of the *partition*
 - A concept of the *view*
 - A concept of the *State Identifier*
 - A global (in the scope of the distributed database) secondary key for CDB contents to resolve an ambiguity when there are two or more objects for the same validity time
- **CDB has brand new API (C++)**
 - To reflect new concepts introduced in design
 - To be (mostly) persistent technology neutral
 - To allow multiple simultaneous implementations (and technologies)

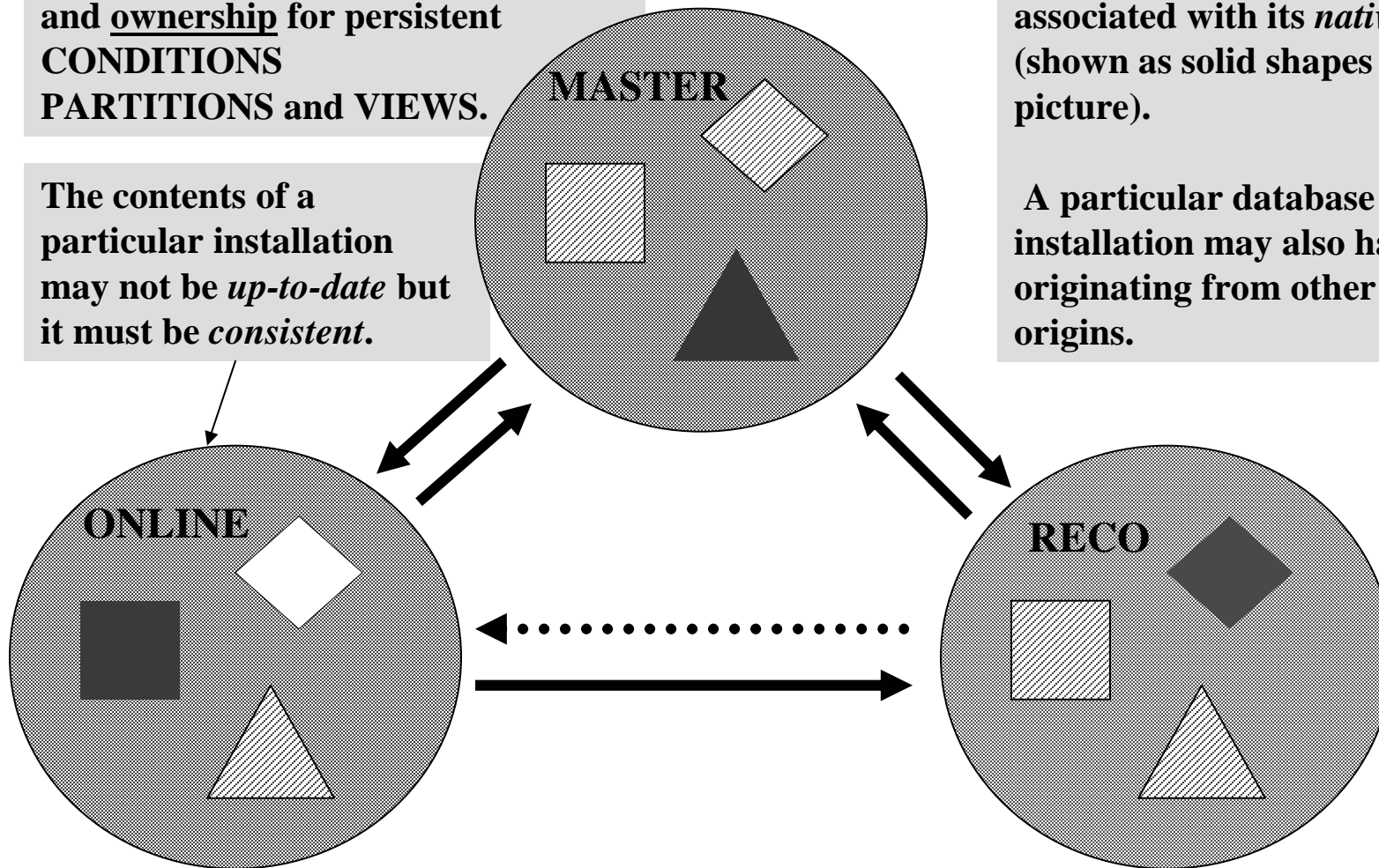
CDB Concepts : Origins

ORIGINS provide a top-level scope and ownership for persistent **CONDITIONS PARTITIONS** and **VIEWS**.

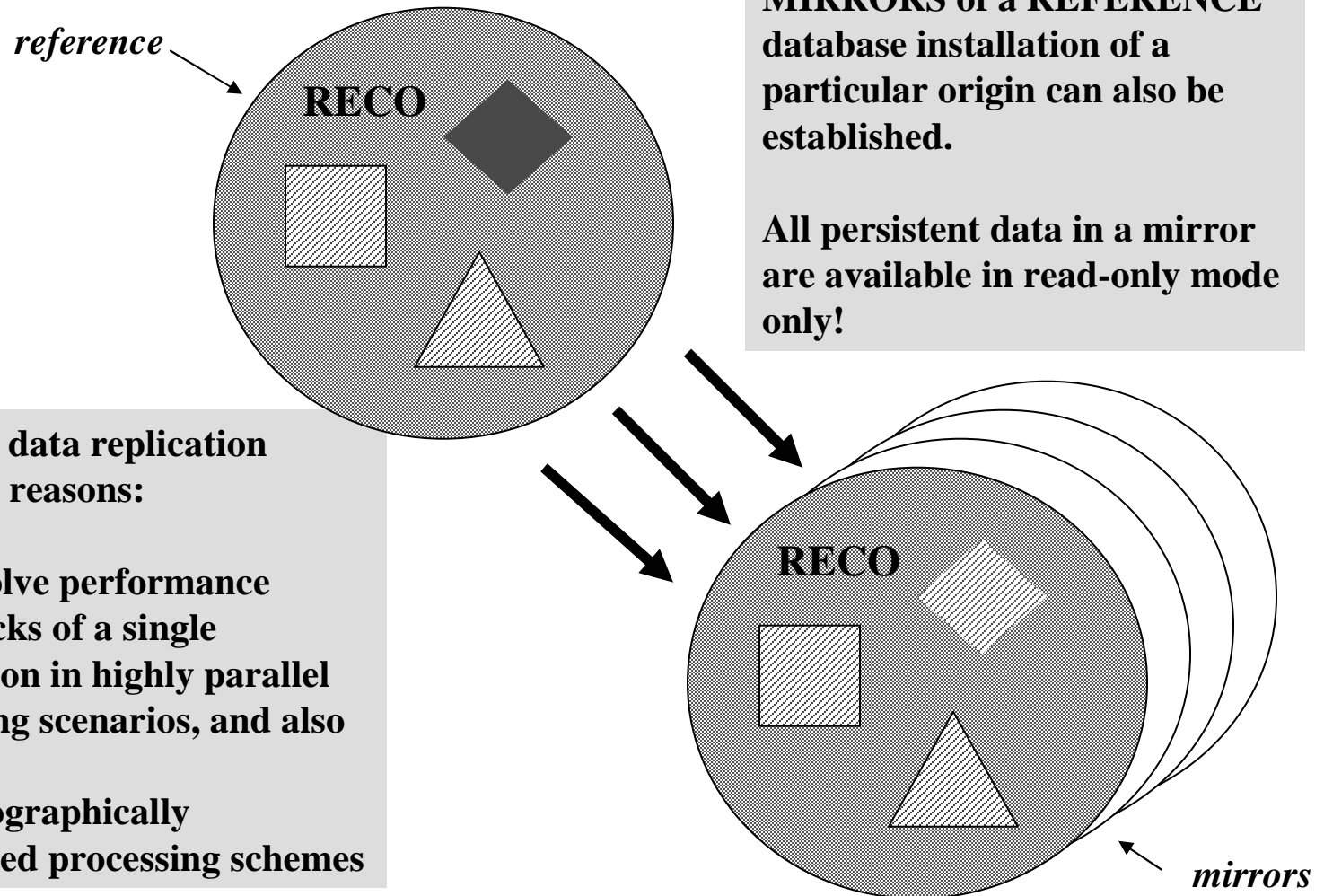
The contents of a particular installation may not be *up-to-date* but it must be *consistent*.

Each database installation is associated with its *native* origin (shown as solid shapes on the picture).

A particular database installation may also have data originating from other (*foreign*) origins.

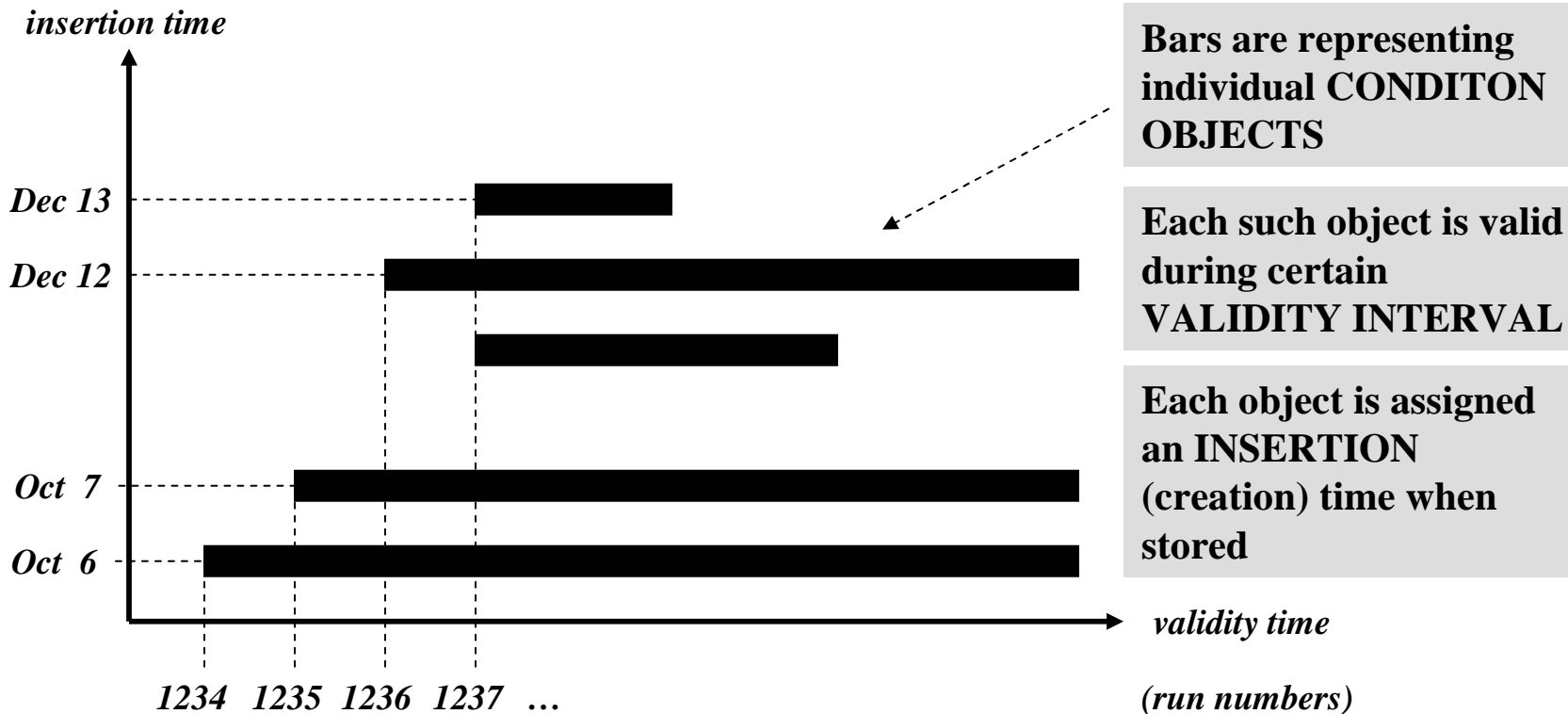


CDB Concepts : Origins : Mirrors



CDB Concepts : 2-D space of conditions (1)

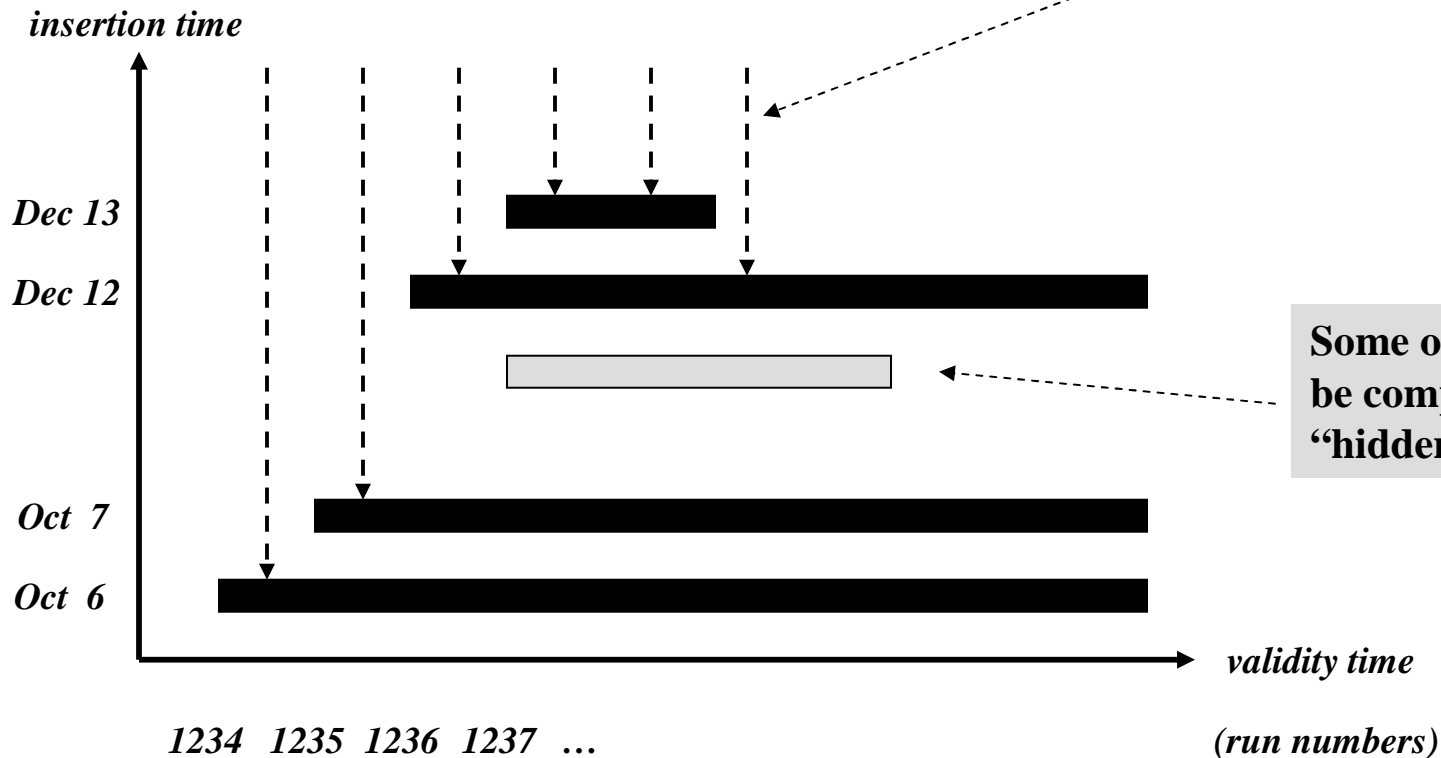
For conditions, CDB introduces a simple geometric model in which conditions are containers providing 2-D space (INSERTION and VALIDITY timelines) for objects.



Run numbers are used in this picture just for simplicity. The actual CDB API and its implementation use a special class representing time: `BdbTime`

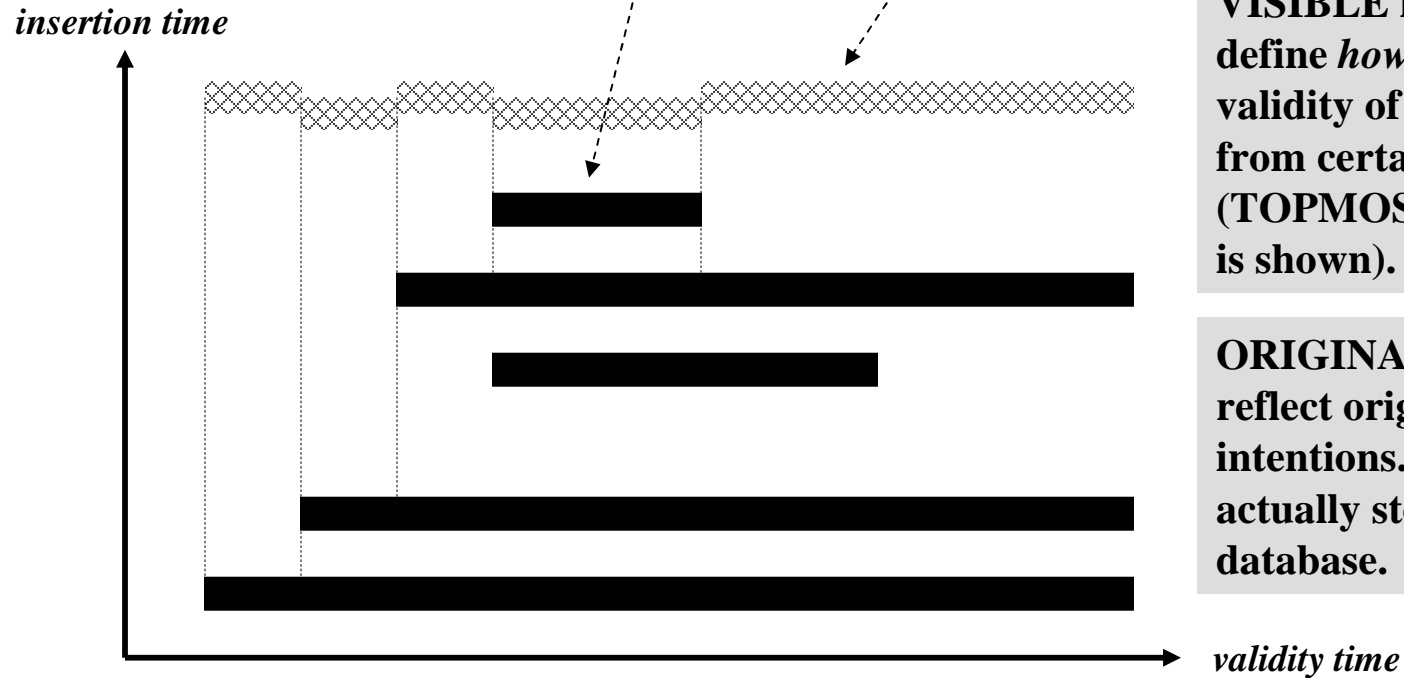
CDB Concepts : 2-D space of conditions (2)

RESOLVING OBJECTS : in this oversimplified example only the most recent conditions “visible” from some point of the INSERTION timeline are seen.



CDB Concepts : 2-D space of conditions (3)

Two types of validity intervals: ORIGINAL and VISIBLE.



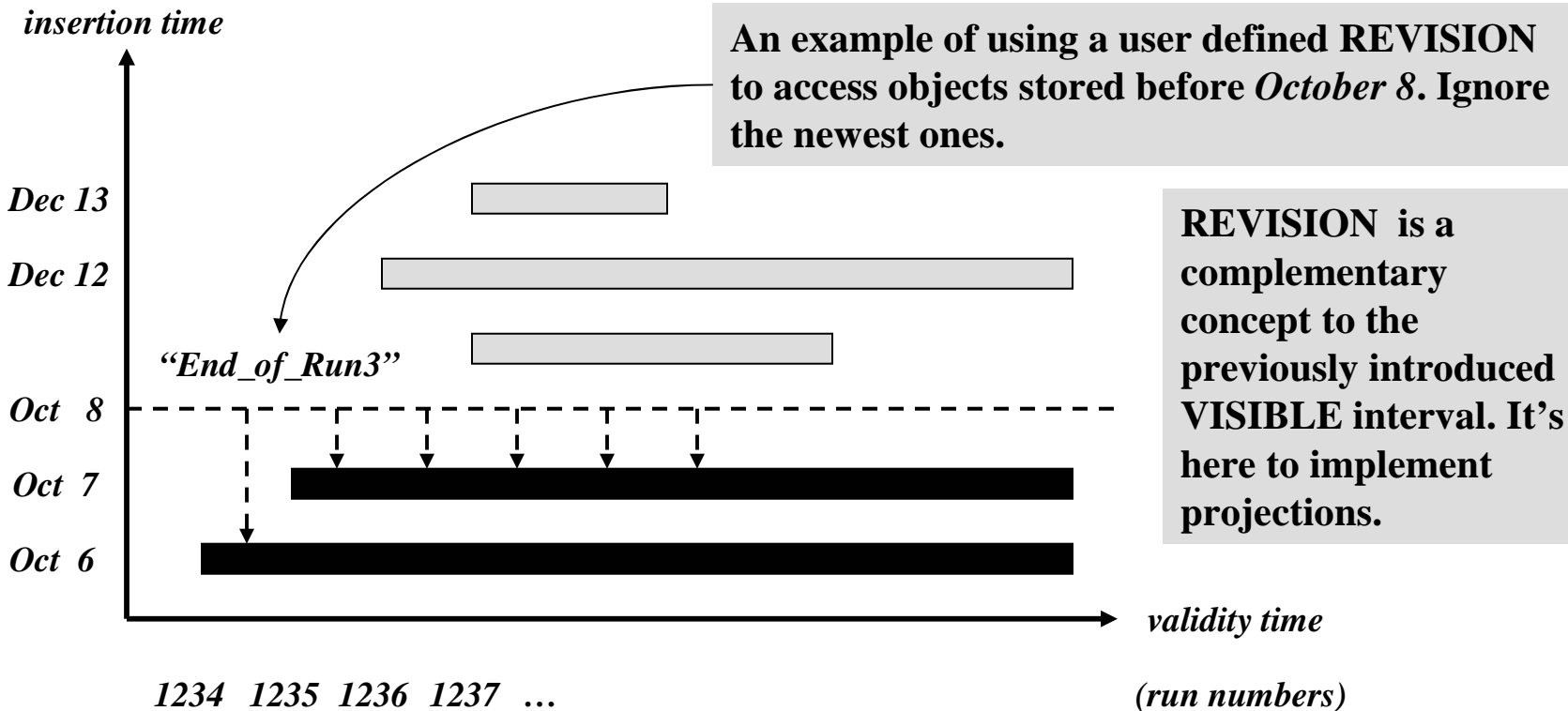
VISIBLE intervals define *how* we see the validity of stored objects from certain projection (TOPMOST projection is shown).

ORIGINAL intervals reflect original user intentions. This is *what* is actually stored in the database.

VISIBLE intervals may not actually exist in the database in a form of persistent structures. These kind of intervals is required by the current conceptual model of CDB.

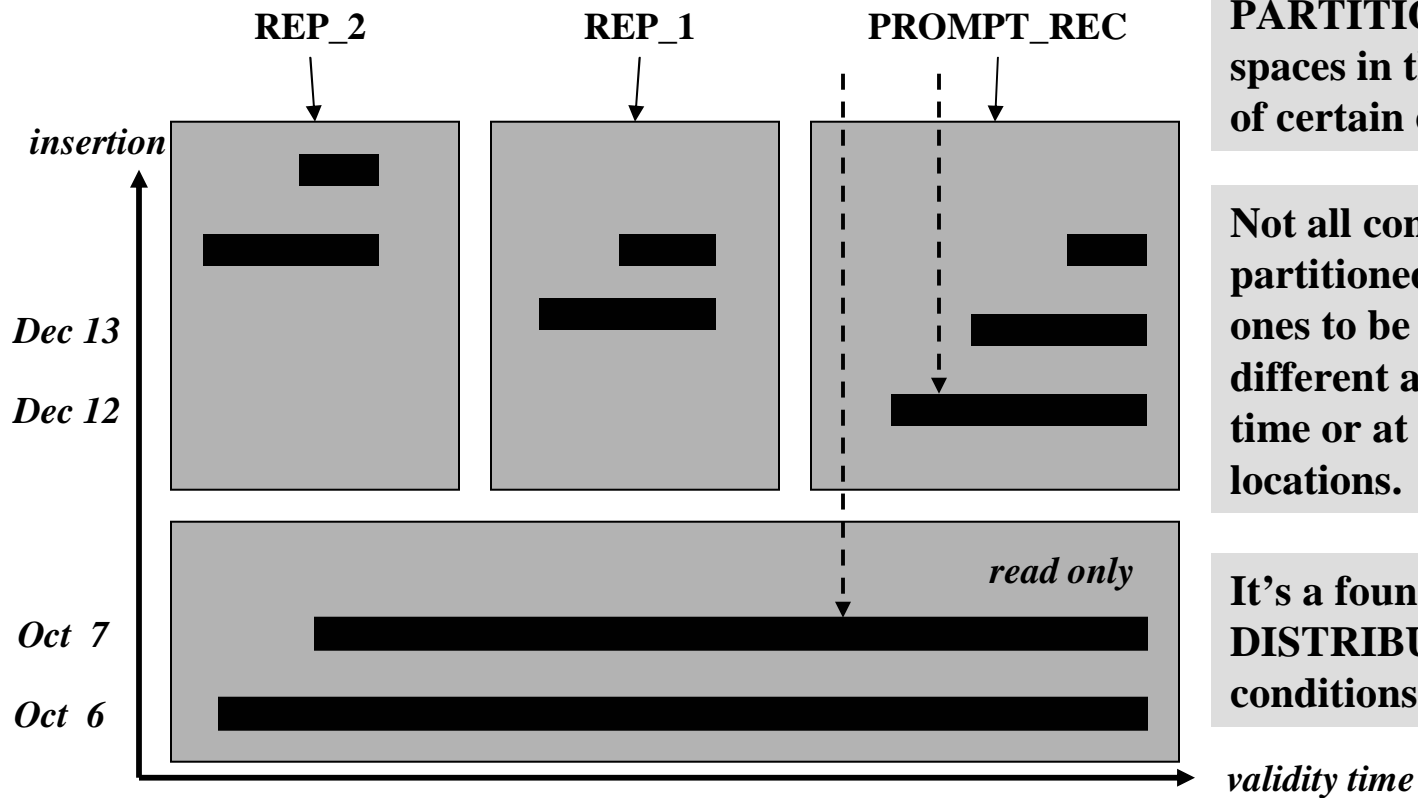
CDB Concepts : 2-D space of conditions (4)

REVISION is a “high-watermark” in the insertion time of a condition separating objects stored *before* from the ones stored *after* certain INSERTION time.



REVISION is identified by its *timestamp* (in the INSERTION timeline) or an arbitrary *name* specified by a user at its creation time.

CDB Concepts : Partitions



PARTITIONS are subspaces 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.

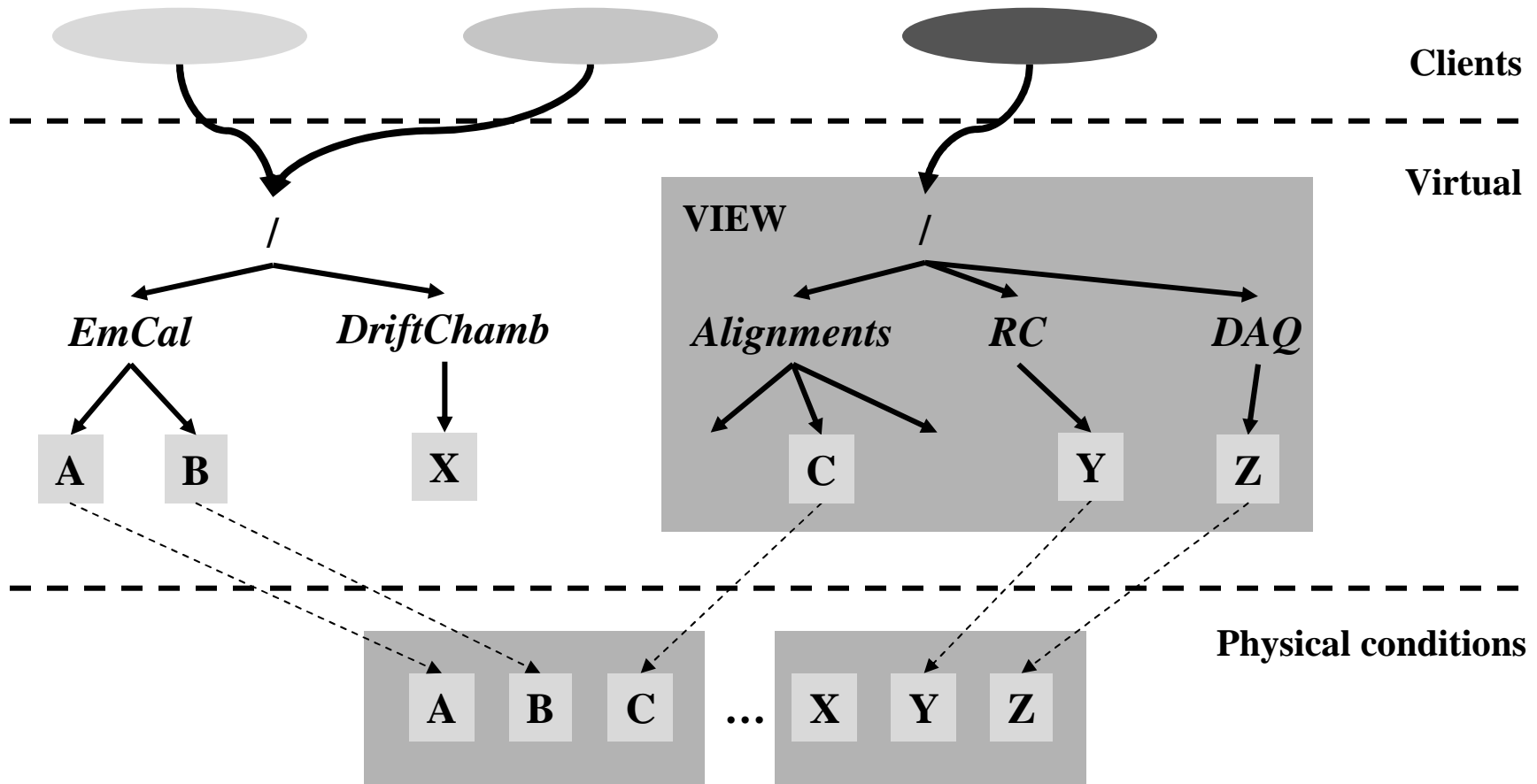
It's a foundation for **DISTRIBUTED** conditions.

Partitions are owned by **ORIGINS**. All storing operations for new objects are confined within the limits of the corresponding partition.

Meanwhile, partitions are transparent to the data access operations.

CDB Concepts : Views : Namespaces

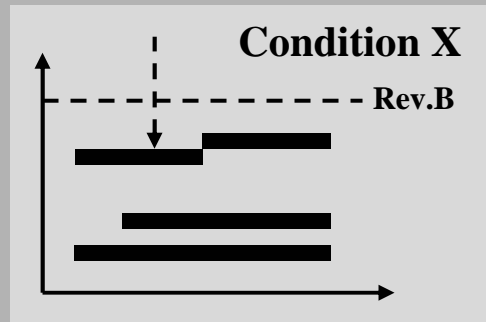
THE GENERAL IDEA: a concept of the **VIEW** lets us to treat the database contents in different ways for various kinds of clients or CDB installations.



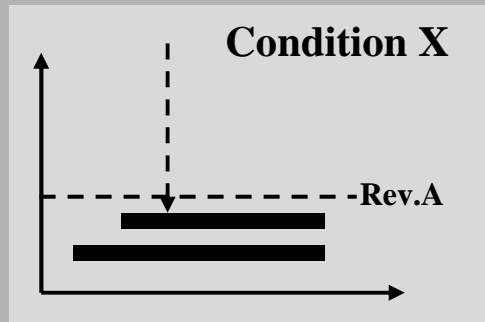
CDB Concepts : Views : Configurations

Each condition in a view has a CONFIGURATION *implicitly* defining a secondary key (REVISION) to be used when resolving intervals in the condition. Views allow to avoid knowing *explicit* secondary keys when there is an ambiguity of multiple objects at the same point of validity time. Views are used as single configuration keys for CDB applications.

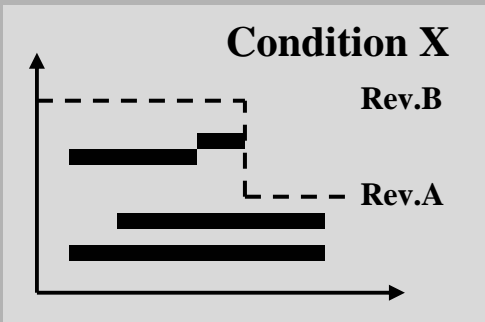
View 1



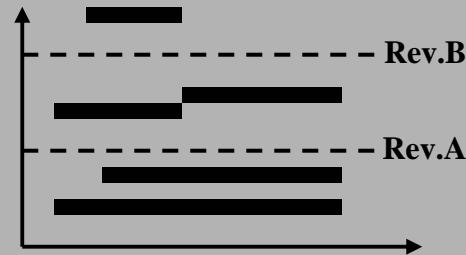
View 2



View 3

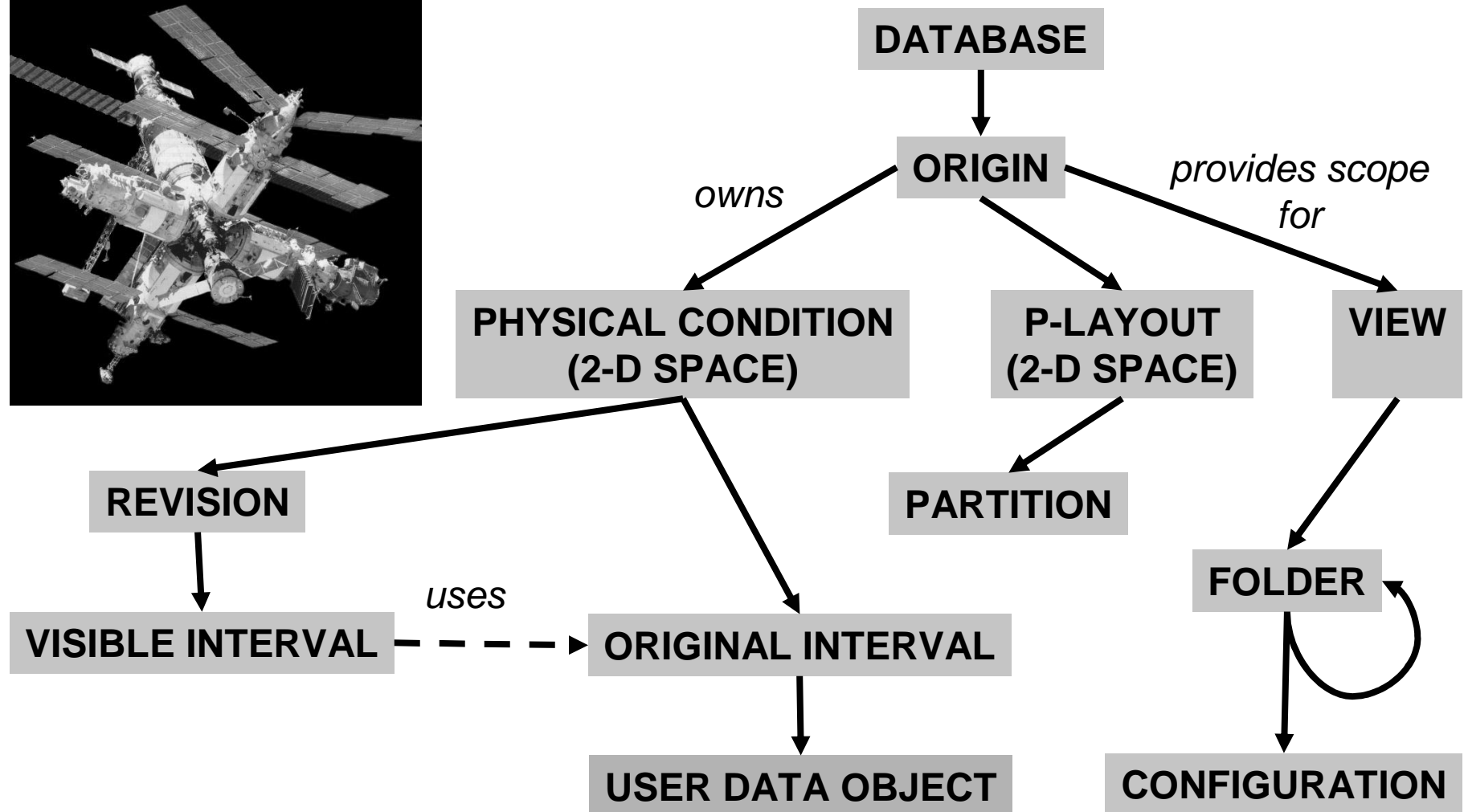
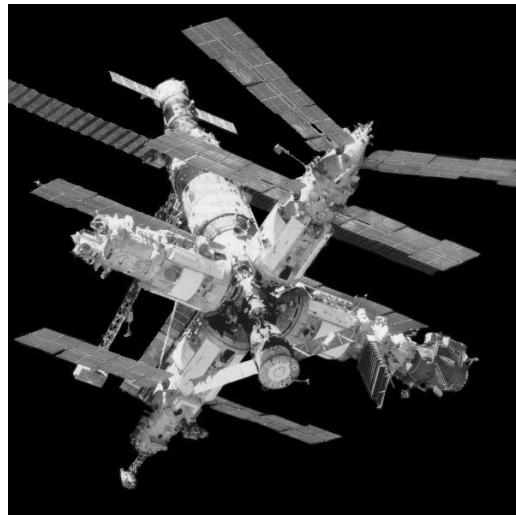


Condition X

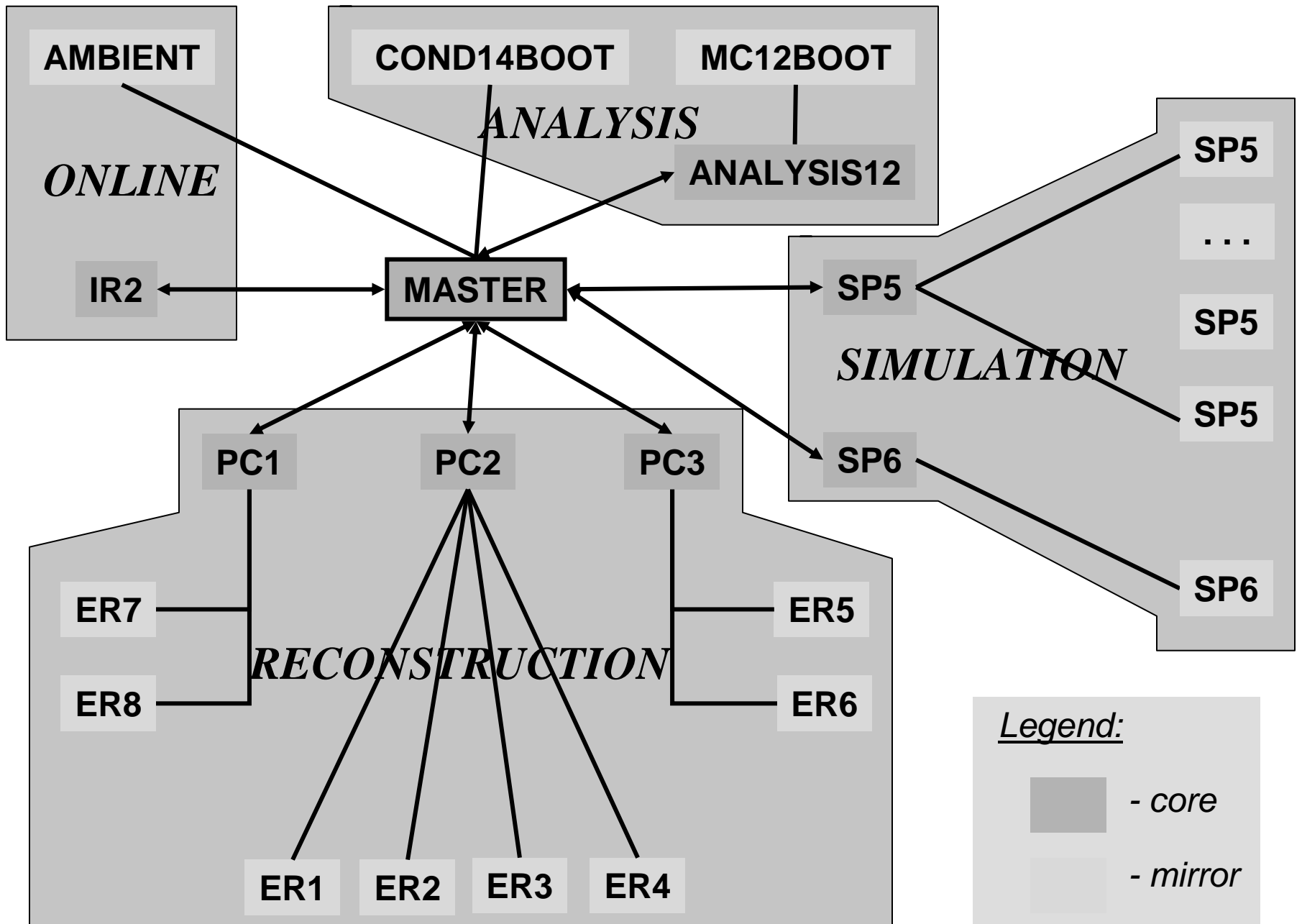


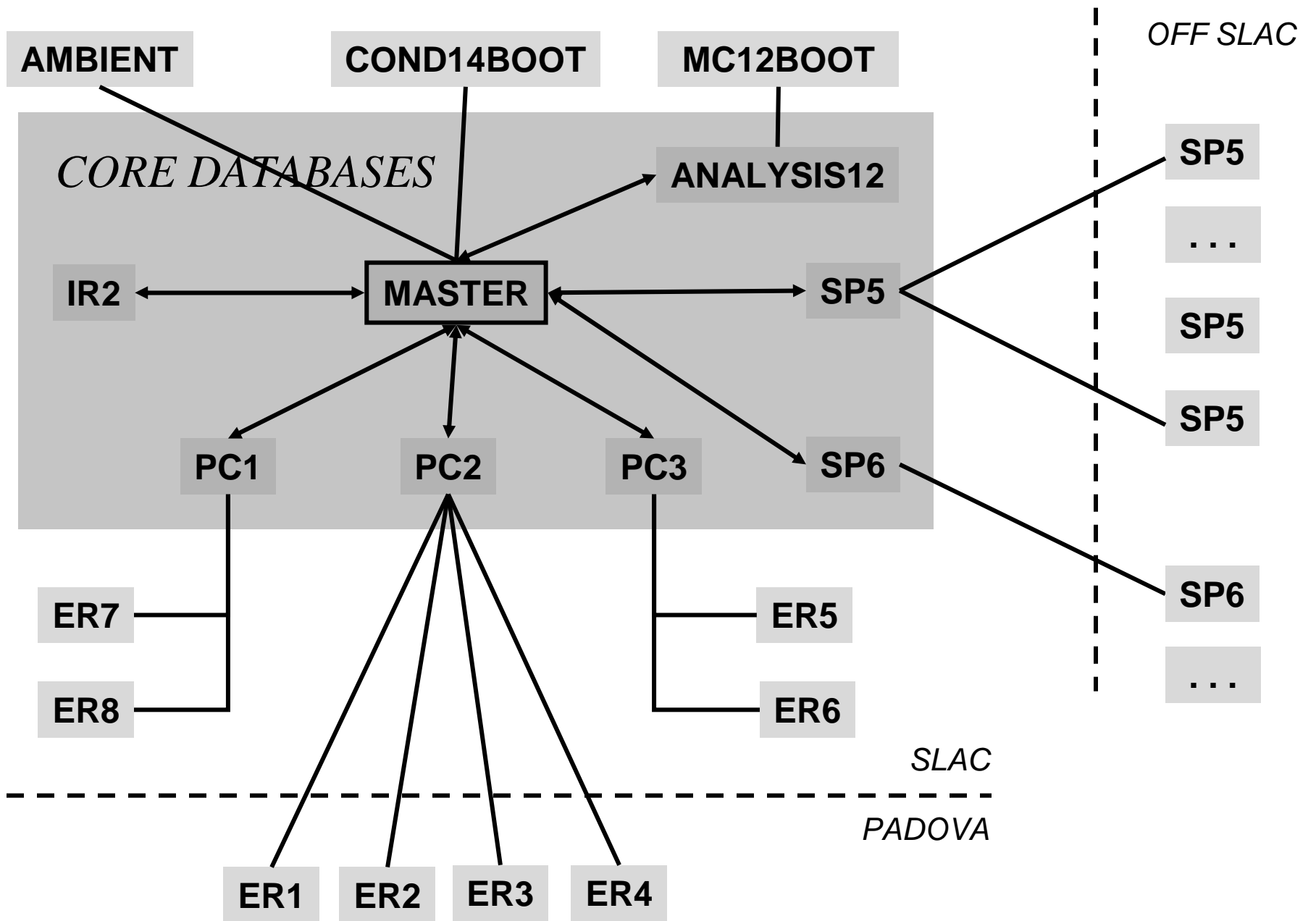
Physical conditions

CDB Concepts : Scope & Ownership Diagram



Using CDB in BaBar...





Current Database Status : Schema

- **Persistent technology: Objectivity/DB (DDL)**
- **Complex model of metadata**
 - Not so many persistent classes though (<50)
 - Not very OO (more like a relational schema)
 - Several non-trivial solutions for performance and efficiency (of the persistent space usage)
- **A variety of user defined payload classes**
 - Over 400 of persistent classes
 - About 20 of them are “composite” ones
 - (Almost) no control on how users do their modeling
 - No common design pattern (except “*cut-and-paste*”)
 - Using OO in “full throttle”(!)
- **Issues**
 - Too many users were involved into persistent technology specific development
 - Too much freedom in a way the classes are defined complicates any migrations
 - Was it really necessarily?
 - Perhaps 95% applications can be solved using “*cookie-cutter*” approach?

Current Database Status : Schema : Tables

- **Added in 2004**
 - A CDB API and persistent schema extension for storing tables (*N-Tuples*) with user data
 - (Practically) unlimited size
 - Only primitive types of elements are supported
 - No need to develop new persistent classes
 - Is good for certain applications only
- **Is used in production**
 - See next slide...
- **Can be easily implemented in other technologies**
- **A similar model can be followed to build other types of predefined containers for user data!**

Current Database Status : Size

- **32 GB as of today**
- **1.5 million of persistent objects with user data**
- **Over 400 of physical files in an Objectivity/DB federation**
- **Nearly 500 various conditions before Summer 2004:**
 - 2/3-rd of them originated in ONLINE
 - 1/3-rd originated OFFLINE
 - 38 of them are so called “rolling calibration”-s
- **About 1400 of new “PID Efficiency Tables” and ”SVT Tracking Tables” conditions put into CDB just recently:**
 - All belong to OFFLINE
 - Use CDB API “**N-Tuple**” extension
 - Contributed nearly 1 GB into the overall size
- **8 origins**
- **40 views**
- **20 partitions**

Issues, Developments, Etc...

- **Major issues**
 - Synchronizing distributed database installations
 - Not a trivial problem (even though CDB has the corresponding provisions in its design)
 - We're still looking into more adequate mechanisms and technologies
 - Managing database contents (*conditions*, *partitions* and *views*)
 - A complex logical model of CDB requires more management work
 - Users also need to be educated in CDB
 - Corresponding procedures and protocols are yet to be finalized
 - A very complex user defined schema
 - It's difficult to migrate the contents of CDB to other persistent technologies
- **Developments...**
 - Are mostly related to BaBar's "Computer Model 2" (CM2)
 - Implementing CDB in:
 - RDBMS (MySQL) and ROOT/CINT objects as BLOB
 - For **read+write** use (central production model for "core" databases)
 - Pure ROOT I/O
 - For **read-only** use (export model for "mirror" databases)
 - Migrating CDB schema and contents from Objectivity/DB into ROOT/CINT
- **Open questions**
 - What we're going to do with CDB in the GRID "era"?

Summary

- **The Conditions database of BaBar went through two major iterations in its history**
- **Important lessons learned in...**
 - Understanding needs of the contemporary HEP experiment
 - Formulating an adequate logical model of the database
 - Designing, implementing and managing a distributed database
 - Coping with performance and scalability challenges
 - Finding a right level of interaction between database managers, conditions developers and users
 - Using Objectivity/DB in the database implementations
- **Most problematic areas are (still) in...**
 - Synchronizing distributed database installations
 - Managing database contents
 - Dealing with a very complex user defined schema
- **CDB is undergoing the persistent technology migration**