```
"BdbCondPathSingleton::loadFromConfigFile() -- info.
    A configuration file describing a revision/owner path was not found.
    Assuming the default behavior of BdbDatabase::fetch() interface --
    the most recent version of the condition data will always be
    fetched from the database."
```

Quoted from "The BaBar Conditions/DB"

Revisions in the Conditions DB

Igor A.Gaponenko

Lawrence Berkeley National Laboratory

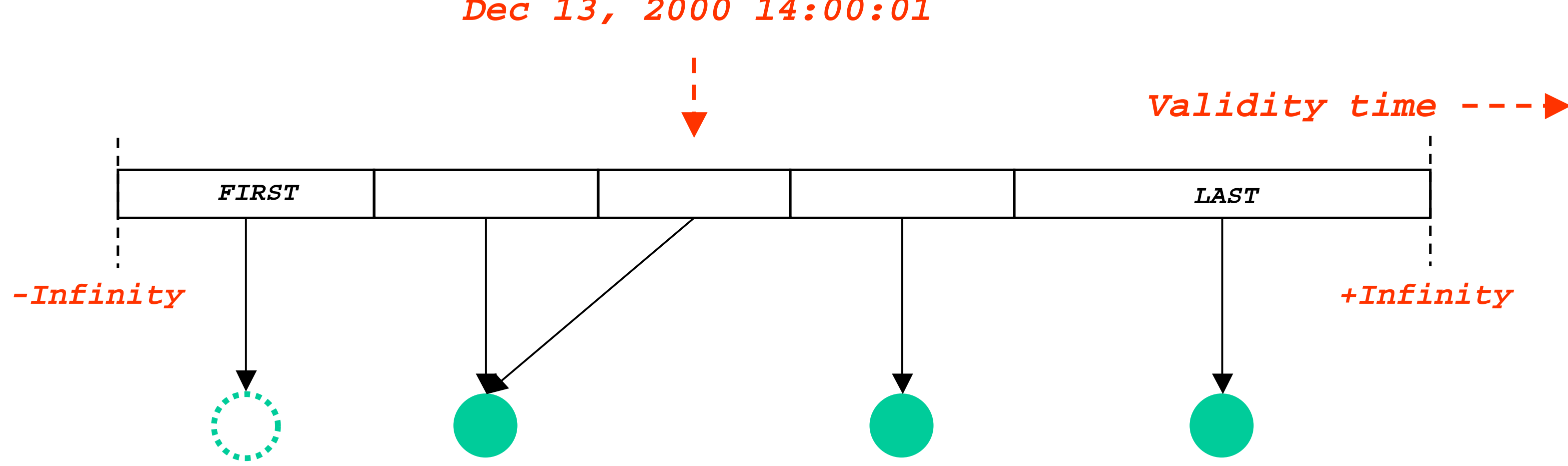
(IAGaponenko@lbl.gov)

On the Agenda…

- The revisions in the Conditions database:
  - Introducing the concept
    - What's a revision?
    - How to use revisions?
    - How to create revisions?
  - Use of revisions in BaBar public federations
    - Which conditions to "revise" and when?
    - The configuration files
  - Related use cases:
    - Creating new revision at the end of reprocessing
    - Revisions for the IR2 conditions
    - Creating new revision when new alignments are added
    - Deleting a revision
    - Etc.
- **BdbIntervalUtil**
  - A general overview of the functionality
  - Special focus on how to deal with revisions
  - **See more information about this tool at:** *http://www.slac.stanford.edu/BFROOT/www/Computing/Offline/Databases/Conditions/Docs/BdbIntervalUtil/SwissArmyKnife.htm*
- Discussion

Meta-data and Conditions Objects

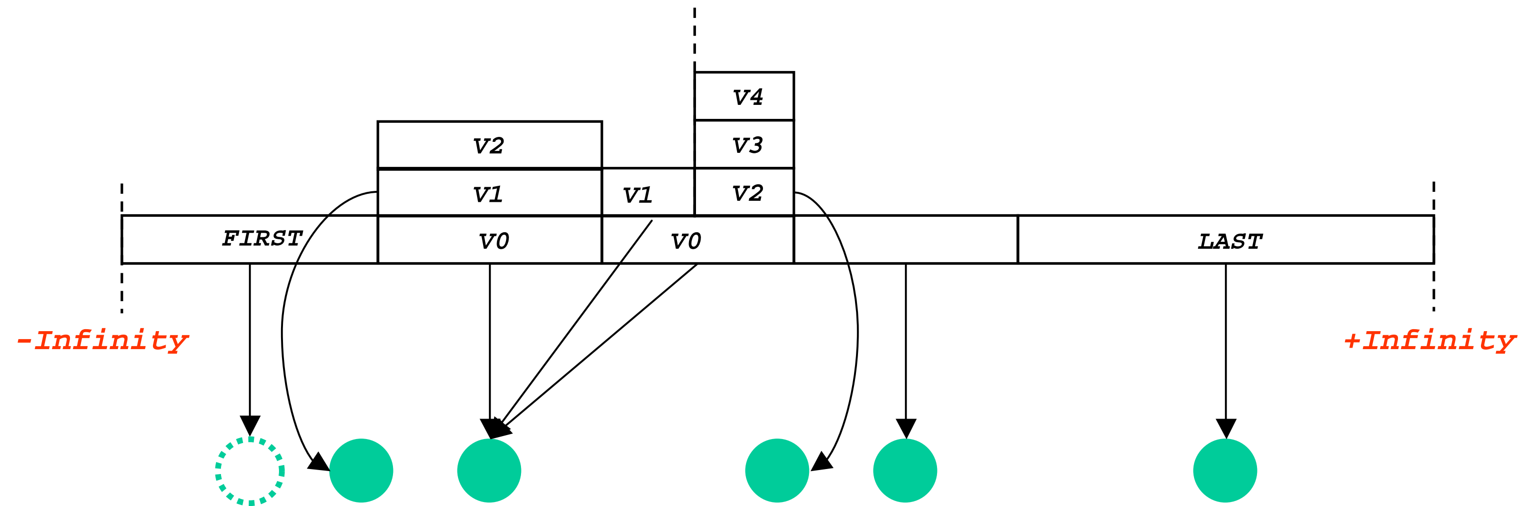
NOTE: *The plain timeline for a condition is shown.*



*The condition is parameterized by a single parameter – **validity time**.*
*The timeline of a condition object is divided onto **intervals**: [begin, end).*
Each interval covers a period in the validity time where the value of the condition is constant.
*The intervals are connected into a linked list. But the Objectivity's **indexing** is used to speed-up a process of locating desired interval for a given time.*
Each interval has pointer to the actual condition object residing in a different database.

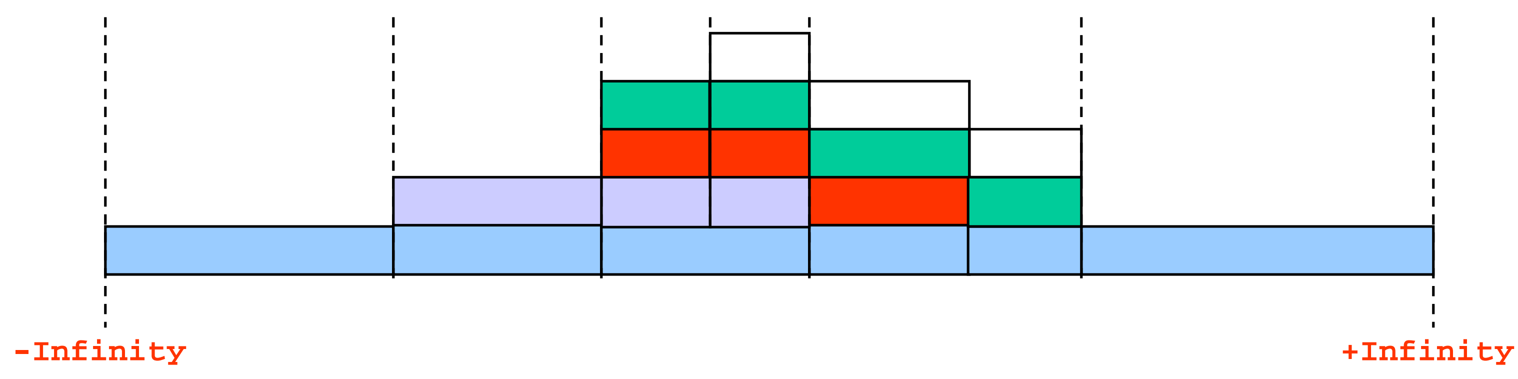
Meta-data and Conditions Objects: versions

NOTE: *The versioned timeline for a condition is shown.*



The intervals above the baseline level are called ***versions****.*
Each interval may have many version within its validity time limits.
The versions are organized into vertical trees.
Each tree grows from a baseline interval.

The Concept of Revisions: Second Dimension



REVISIONS:

<3> = <2> +

<2> = <1> +

<1> = <0> +

<0> = "baseline"

COMMENTS:

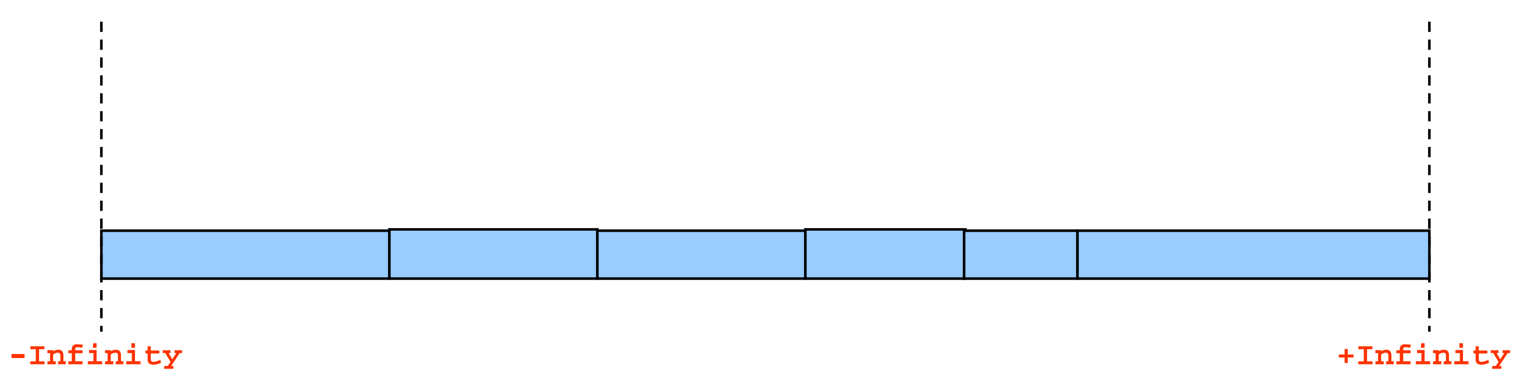
- *A **revision** is a second (vertical) key used when a condition is being fetched.*
- *A revision is made of intervals. Each revision, except a **baseline** one, has a **base** revision. A revision (with its direct and indirect ones) covers the whole timeline.*
- *Each revision with their direct and indirect base revisions provides a complete timeline for the condition.*

The definition of a revision

- ***Revisions*** are uniquely identified by 32-bin unsigned long numbers called ***revisions id-s***.
- A revision **N** is defined as a patch to the revision **M**, where **M** < **N**. The revision number **M** is called a ***base revision*** for the revision **N**.
- A revision may only have one revision as its base.
  - **EXCEPTION**: Revision number **0** is a special one – it's so called ***baseline*** revision. Unlike other revisions this one has no ***base*** revision. All other revisions are either direct or indirect derivatives of the ***baseline*** revision.
- Therefore revisions form a tree growing from the ***baseline*** revision.
- Each revision together with its base revisions (direct and indirect ones) covers the whole timeline of a conditions ranging from logical **–oo** till the logical +**oo**.
- Each condition has its own set of revisions.
- See examples illustrating this concept on the next slides…

Example 1

NOTE: *There is just the* ***baseline*** *revision in a container.*



REVISIONS:

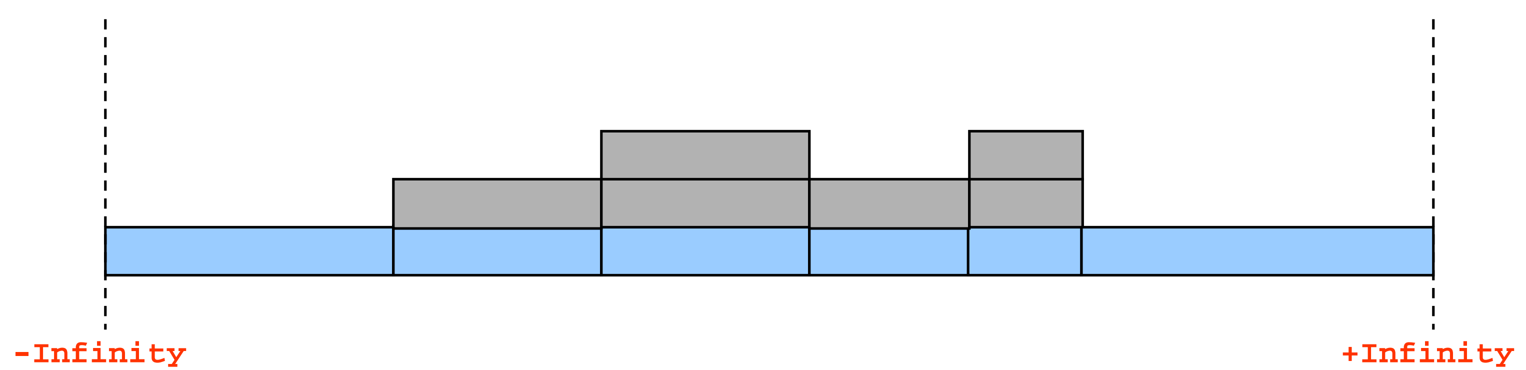
<0> = "baseline"

COMMENTS:

- *The* ***baseline revision*** *is created automatically. Every single interval at this layer is associated with this revision.*
- *The* ***baseline revision*** *has no* ***base*** *revision. Therefore all intervals directly associated with this revision cover complete validity timeline.*
- *This revision can't be destroyed.*

Example 2

NOTE: *There is **baseline** revision and new **unassigned** (to any particular revision) intervals.*



REVISIONS:

"unassigned intervals"

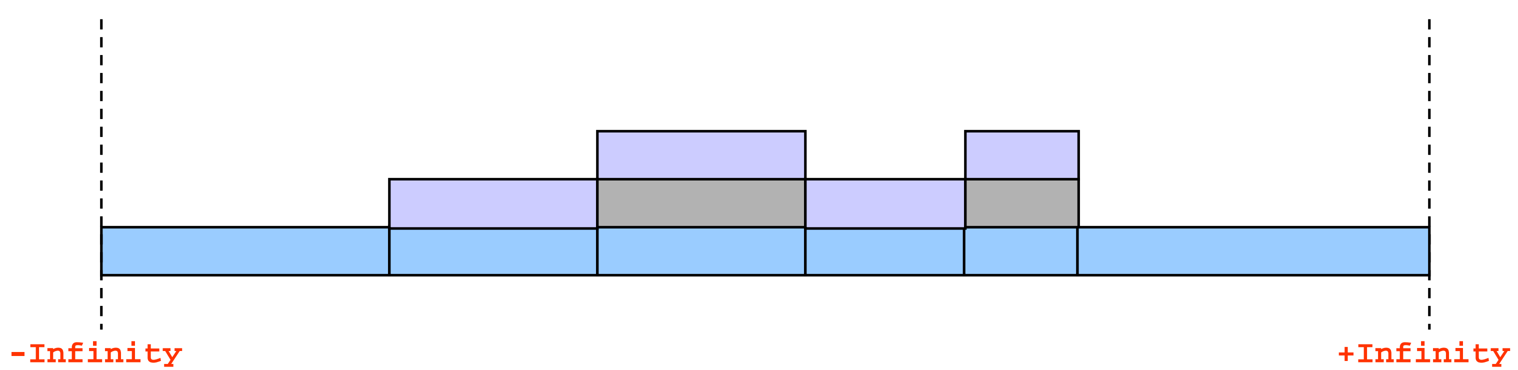
<0> = "baseline"

COMMENTS:

- *The **unassigned intervals** do not belong to any particular revisions.*
- *The only way to access them is when they are on the top of the timeline. In this case we call these intervals the **topmost intervals**.*

Example 3

NOTE: *Revision 1 is based on the* ***baseline*** *revision. It's made of unassigned intervals from the* ***topmost*** *layer.*



REVISIONS:

`<1> = <0> +` [light purple box]

[grey box] "unassigned intervals"

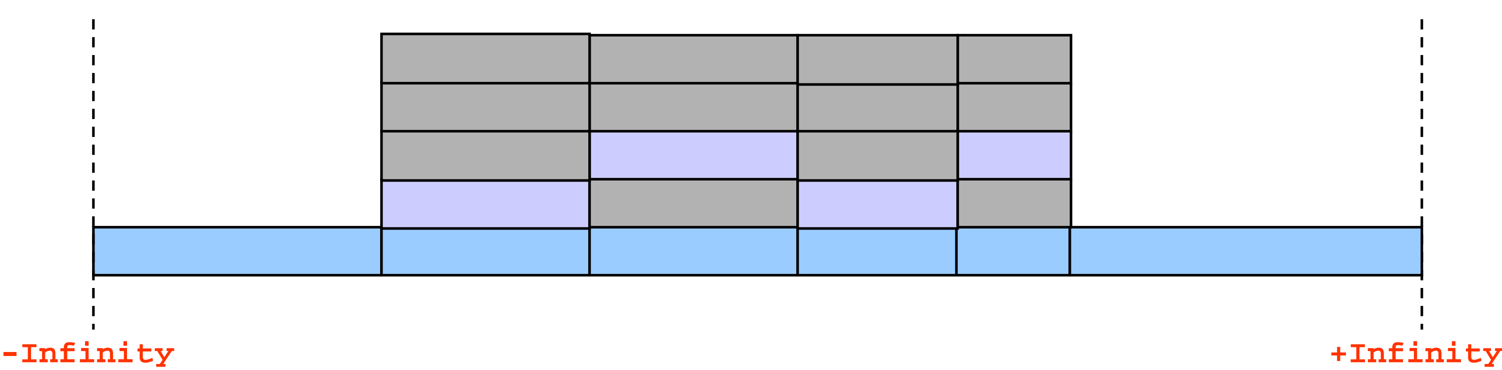
`<0> =` [light blue box] "baseline"

COMMENTS:

- *We still have* ***unassigned intervals*** *in the depth of the genealogy. But they are not accessible anymore.*
- *A new revision 1 directly covers just a fraction of the timeline. But since it's based on the revision 0 then the whole timeline is effectively covered. For example when someone is fetching a revision 1 condition outside directly covered area then what he/she actually gets is a revision 0 intervals.*

Example 4

NOTE: *More **unassigned** intervals on the top of revision 1.*



REVISIONS:

`<1> = <0> +`

"unassigned intervals"

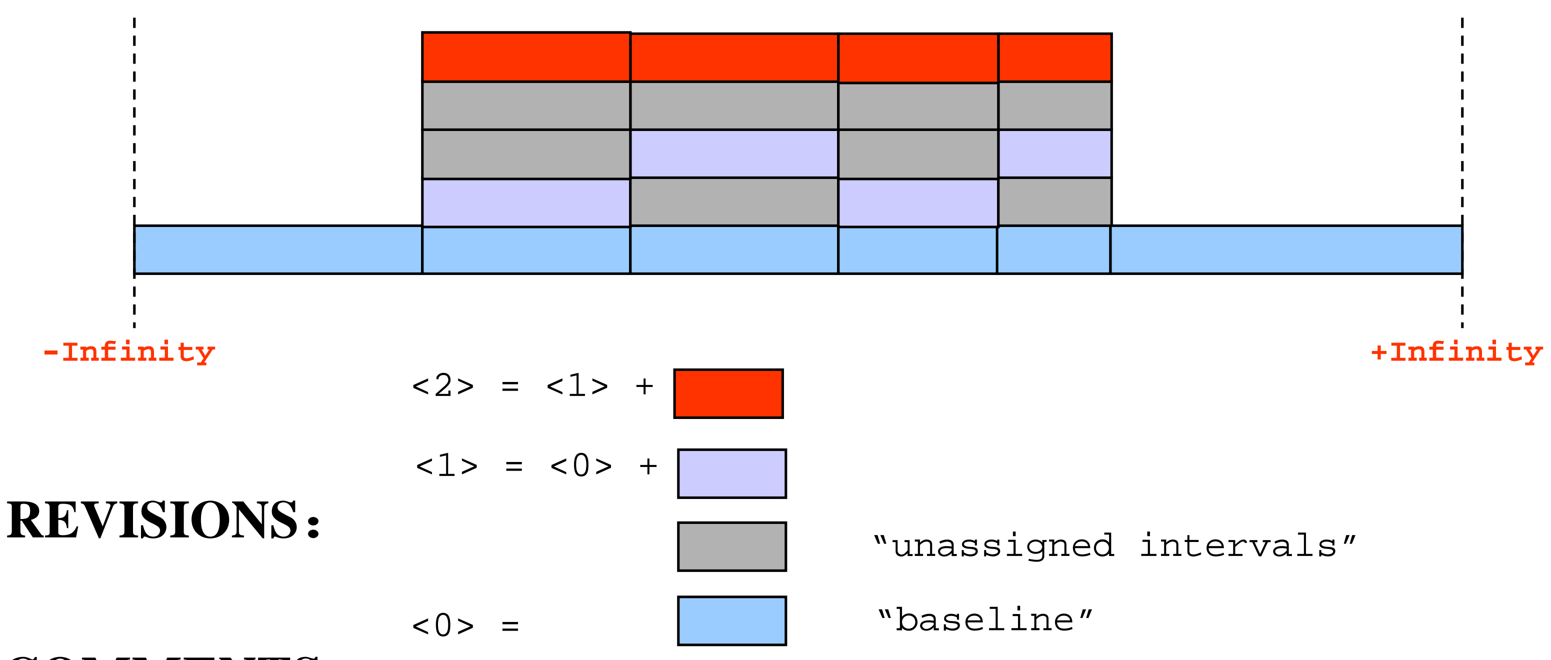
`<0> =` "baseline"

COMMENTS:

- *We have more **unassigned intervals** on the top of the genealogy. Only the topmost intervals can be accessed.*
- *A new revision 1 is deep inside the genealogy, but it's still available unlike other unassigned intervals.*

Example 5

NOTE: *New revision 2 is based on revision 1.*



REVISIONS:

COMMENTS:

- *We created new revision overriding the previous one.*
- *We still have* ***unassigned intervals*** *in the depth of the genealogy.*
- ***NOTE****: This example is also meant to illustrate the semantics of* ***base*** *revision, which means "a revision is a patch to its base one", but not necessarily that "a base revision was used to produce conditions in a derived one".*

Using revisions

- Problem:
  - The legacy API does not support revisions directly as a second key. Most of our code was created *before* the revisions were introduced into Conditions DB API.
    - May 1999
  - See the next 2 slides for the examples of typical code…
- Solution:
  - Revisions are configured externally via a singleton class: **BdbCond/BdbCondPath**
  - The dictionary of the singleton is loaded by any (or both) of the following means:
    - Implicitly from a ***configuration file***
      - First the following environment variable ***BDBCOND_PATH*** is checked.
      - Then if it's not found then the following file is expected: ***$CWD/conditions.cfg***
      - If both methods above fail then a message is issued and topmost conditions are used.
    - Explicitly using an API of the singleton
      - This method allows to override the implicit loading described above
- The syntax of the Configuration file
- The API of the singleton

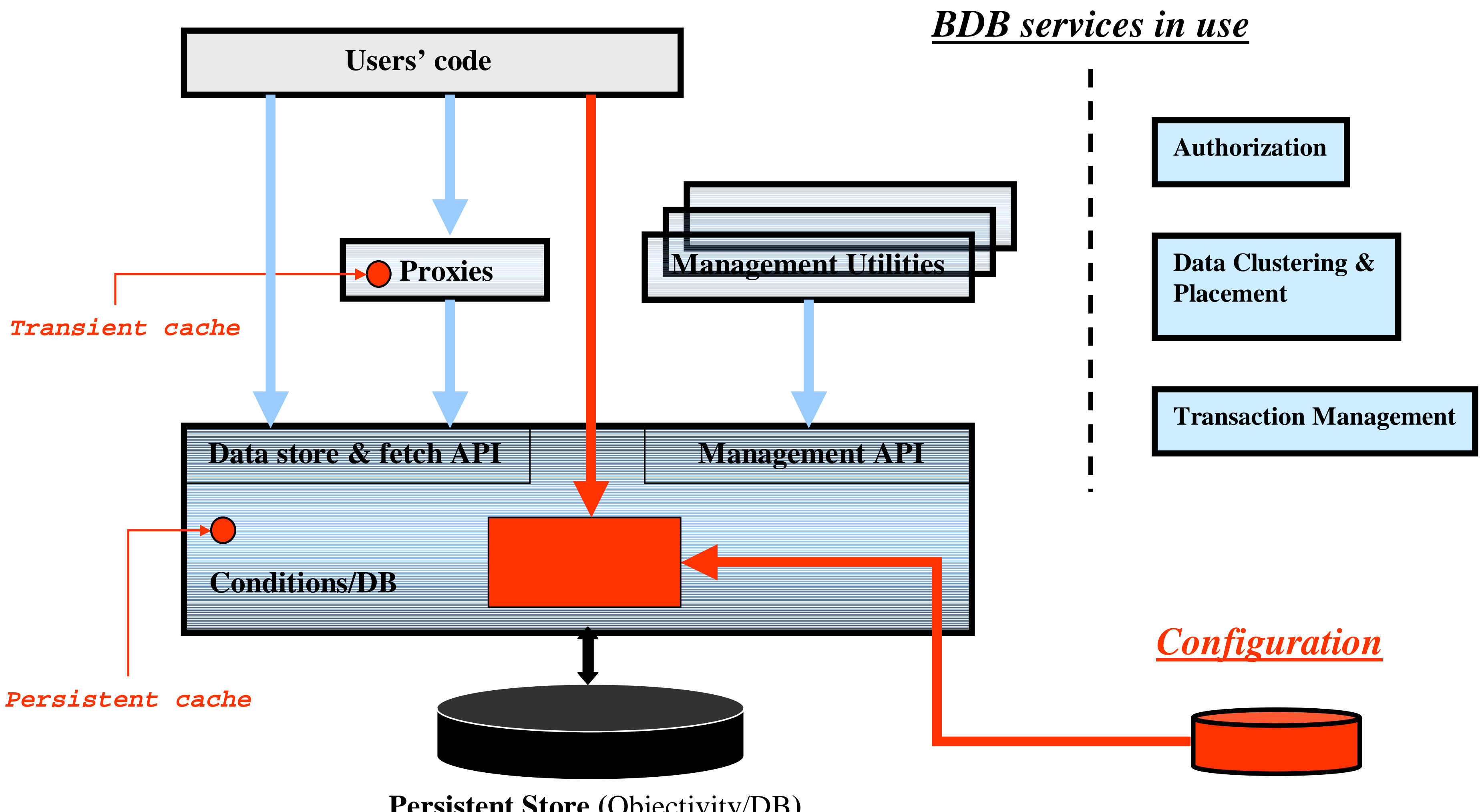
An Example of the Conditions/DB Use: Storing

```
#include “BdbClustering/BdbCondClusteringHint.hh
#include “UserPackage/MyObject.hh”
#include “BdbTime/BdbTime.hh”
#include “BdbCond/BdbDatabase.hh”
#include “BdbCond/BdbObject.hh”

// Create a new persistent object at given location

BdbCondClusteringHint theHint( “emc” );
BdbHandle(BdbObject) theH = new( theHint.updatedHint( )) MyObject( );

// Register a new object in the database (create meta-data).

BdbDatabase theDb( “emc” );
BdbTime theBegin = 1;
BdbTime theEnd   = 10;

BdbStatus status;
status = theDb.store( theH, “MyClass”, theBegin, theEnd );
if( BdbcSuccess != status ) {
}
```

An Example of the Conditions/DB Use: Fetching

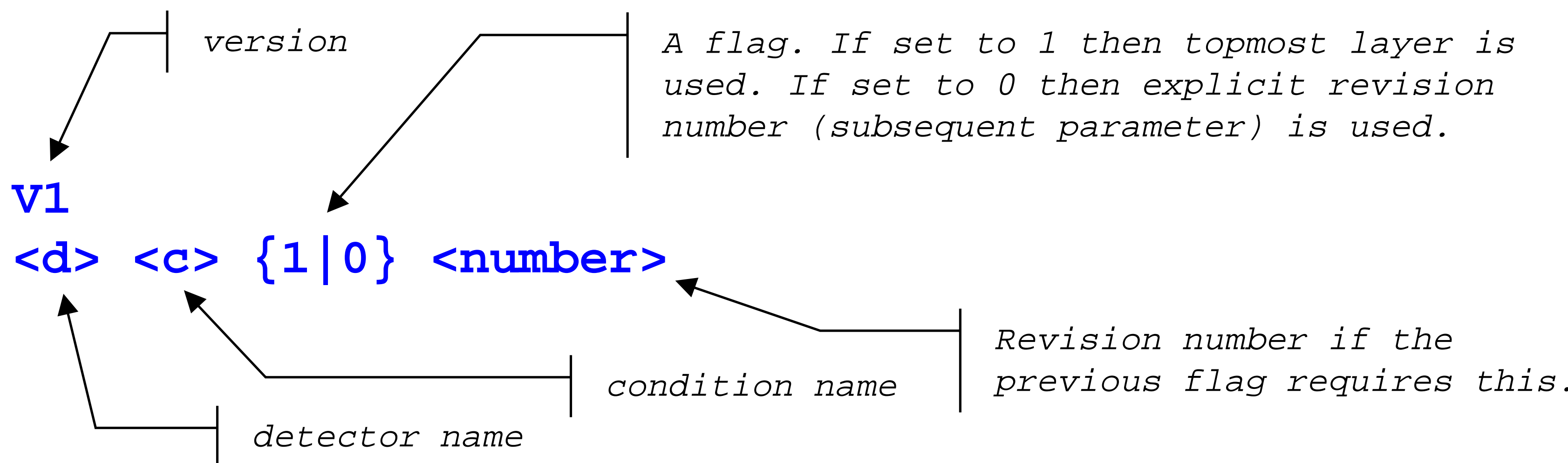
```
#include "UserPackage/MyObject.hh"
#include "BdbTime/BdbTime.hh"
#include "BdbCond/BdbDatabase.hh"
#include "BdbCond/BdbObject.hh"

// Find an object for given criteria

BdbDatabase theDb( "emc" );
BdbHandle(BdbObject) theH;
BdbTime theTime = 1;

BdbStatus status;
status = theDb.fetch( theH, "MyClass", theTime );
if( BdbcSuccess != status ) {
}

// Cast to a concrete class

BdbHandle(MyObject) myH;
myH = (BdbHandle(MyObject)) theH;
```

Two ways to setup the Configuration of Revisions



The Syntax of the Configuration file: v1

SYNTAX:



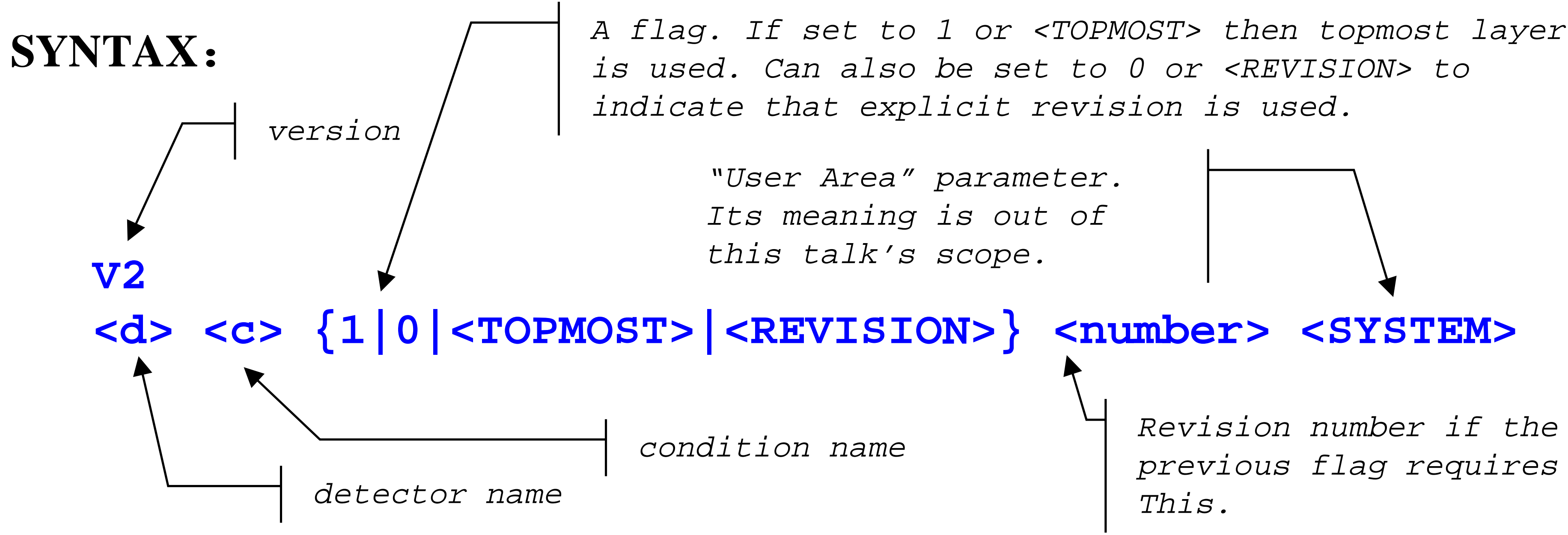
EXAMPLE:

```
V1
*   *             1 0
emc *             0 3
emc EmcPulserCal  1 0
```

- ***use specified revisions for specified containers only.***
- ***the topmost intervals are used for all other containers***

The Syntax of the Configuration file: v2

SYNTAX:



EXAMPLE:

```
v2
*   *            <TOPMOST>  0 <SYSTEM>
emc *            <REVISION> 3 <SYSTEM>
emc EmcPulserCal <TOPMOST>  0 <SYSTEM>
```

BdbCond/BdbCondPath

API:

```
Class BdbCondPath {
public:

  // It's a singleton class
    static BdbCondPath* instance( );

  // Accessors
    BdbStatus getRevision( const char* theDetectorName,
                           const char* theContainerName,
                           bool&       useTopVersionFlag,
                           d_ULong&    theRevisionId );

  // Modifiers
    BdbStatus setRevision( const char* theDetectorName,
                           const char* theContainerName,
                           bool        useTopVersionFlag,
                           d_ULong     theRevisionId );

  // Operations
    BdbStatus loadConfiguration(  const char* theFileName );
    BdbStatus mergeConfiguration( const char* theFileName );
    ...
};
```

Reload/load configuration from specified file. (annotates `loadConfiguration`)

Add configuration from specified file. (annotates `mergeConfiguration`)

Creating a new revision

- Implications:
  - The configuration file has to be updated as well.
- **Step I**: What's required:
  - Intervals
    - Typically the most recent ones from the *topmost* layer
    - More sophisticated search techniques also exist
  - A base revision number
    - Typically the one created before (a linear list of revisions)
    - Or any existing revision (a tree of revisions)
- **Step II**: Instruments
  - **BdbIntervalUtil**
    - Supports a number of ways to browse the contents of interval containers (*genealogy*, *history*, *revisions*) as well as revision creation tools.
      - See "*The BdbIntervalUtil Reference Manual*" for details.
  - Low-level API provided by **BdbCond/BdbDatabase** class.
    - It's more difficult in use. Must be considered in non-standard cases only.

Creating a new revision: BdbIntervalUtil

“revisetop”: *create new revision based on unassigned topmost intervals.*

```
BdbIntervalUtil revisetop <detector> <container> <base_revision_id>
                [-description <description>]
                [-begin_validity <time> [-begin_validity_tzone <tzone>]]
                [-end_validity   <time> [-end_validity_tzone   <tzone>]]
```

“revise2d”: *search intervals in 2-D space of **validity** and **version** time then create new revision.*

```
BdbIntervalUtil revise2d <detector> <container> <base_revision_id> <strategy_key>
                [-description <description>]
                [-begin_validity <time> [-begin_validity_tzone <tzone>]]
                [-end_validity   <time> [-end_validity_tzone   <tzone>]]
                [-begin_version  <time> [-begin_version_tzone  <tzone>]]
                [-end_version    <time> [-end_version_tzone    <tzone>]]
```

“reviseoids”: *create new revision based on a list of intervals’ OID-s.*

```
BdbIntervalUtil reviseoids <detector> <container> <base_revision_id> <oids_file>
                [-description <description>]
```

Creating a new revision: BdbCond/BdbDatabase

API:

```
Class BdbDatabase ... {
public:

  // Genealogy browsing methods…

    ...

  // Create new revision from intervals...
    BdbStatus
    createRevision( RWTValOrderedVector<BdbHandle(BdbInterval)>& theList,
                    const char*                                  theContainerName,
                    d_ULong&                                     theRevisionId,
                    d_ULong                                      theBaseRevisionId,
                    const char*                                  theSite = (char*)0,
                    const char*                                  theDescr = (char*)0 );

  // Inquire about "recent" revisions...
    d_ULong getMostRecentRevisionId( const char* theContainerName );
    d_ULong getNextAvailableRevisionId( const char* theContainerName );
    ...
};
```

Updating an existing revision

- This may happen when more intervals are decided to be associated with existing revision.
  - One of the cases would be when a revision has to be extended toward logical **+oo** as new conditions were added.
- Implications:
  - The configuration file does not change.
  - The response of the Conditions/DB code – does!
- **Step I**: What's required:
  - Unassigned intervals
    - the same as when a new revision is created
      - **NOTE**: They should not overlap in validity time with revised intervals of the same number which is supposed to be updated.
  - An existing revision number
- **Step II**: Instruments – the same as when a new revision is created.
  - **BdbIntervalUtil**
  - Low-level API provided by **BdbCond/BdbDatabase** class.

Updating an existing revision: BdbIntervalUtil

"revisetop_update": *update a revision using unassigned topmost intervals.*

```
BdbIntervalUtil revisetop_update <detector> <container> <revision_id>
                [-begin_validity <time> [-begin_validity_tzone <tzone>]]
                [-end_validity   <time> [-end_validity_tzone   <tzone>]]
```

"revise2d_update": *search intervals in 2-D space of* ***validity*** *and* ***version*** *time then update a revision.*

```
BdbIntervalUtil revise2d_update <detector> <container> <revision_id> <strategy_key>
                [-begin_validity <time> [-begin_validity_tzone <tzone>]]
                [-end_validity   <time> [-end_validity_tzone   <tzone>]]
                [-begin_version  <time> [-begin_version_tzone  <tzone>]]
                [-end_version    <time> [-end_version_tzone    <tzone>]]
```

"reviseoids_update": *update a revision using on a list of intervals' OID-s.*

```
BdbIntervalUtil reviseoids_update <detector> <container> <revision_id> <oids_file>
```

Updating an existing revision: BdbCond/BdbDatabase

API:

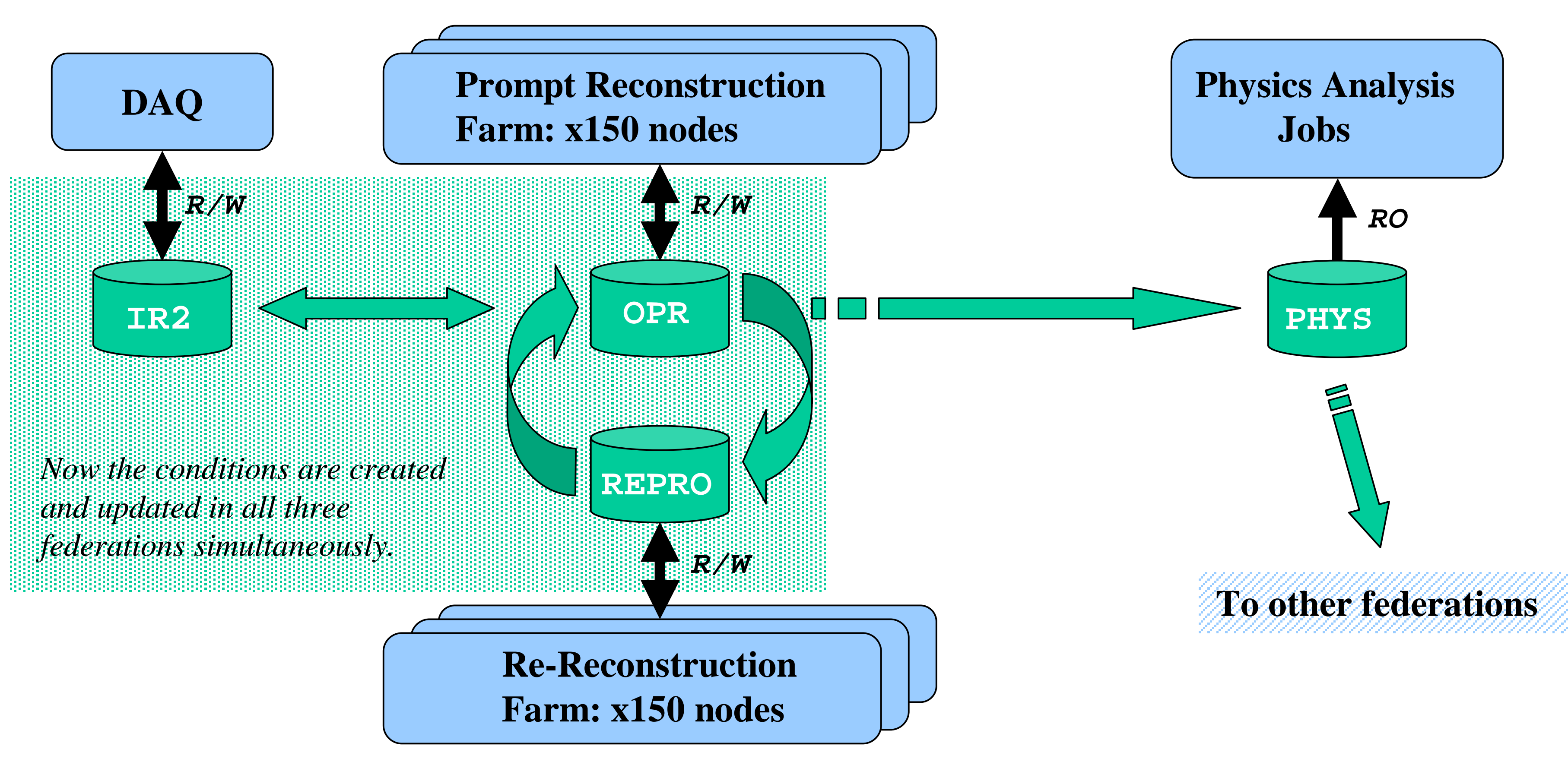
```
Class BdbDatabase ... {
public:

  // Update a revision from intervals...
    BdbStatus
    updateRevision( RWTValOrderedVector<BdbHandle(BdbInterval)>& theList,
                    const char*                                    theContainerName,
                    d_ULong&                                       theRevisionId );
    ...
};
```

The Conditions/DB Setup at SLAC

- (...next 4 slides…)
- The Conditions/DB distributed into 3 federations
- Types of conditions
  - Update frequency varies
  - Different ways to load (automatic or manual)
- Data synchronization procedures:
  - “sweep”
  - “merge”
  - propagate new types of conditions
- Some statistics:
  - A distribution of conditions between detectors

The Federation Setup at SLAC

Distributed Conditions: *All conditions are loaded/updated in 3 federations. The "Rolling Calibration".*



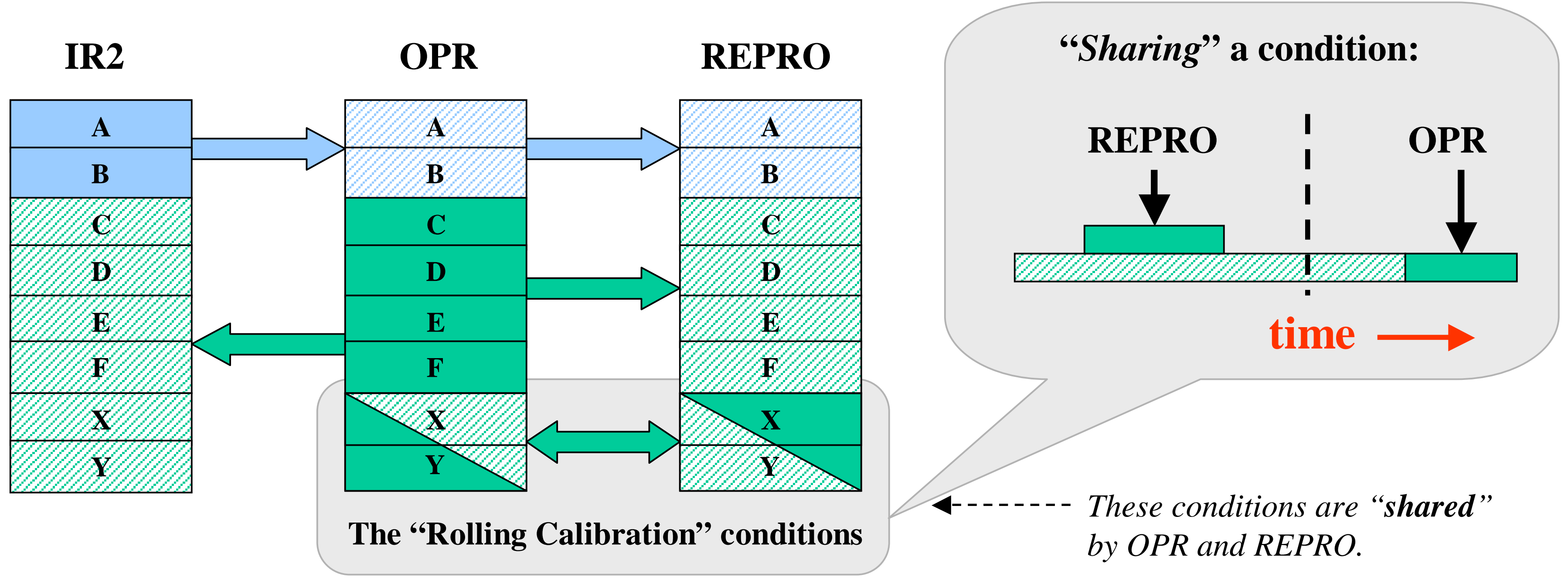
The Federation Setup at SLAC: Types of Conditions

- The role of each “production” federation (the type of data it owns/generates):
  - IR2 Federation
    - online calibrations
    - other conditions under which the events are taken
  - OPR Federation
    - detector alignments
    - original “Rolling Calibration” constants
  - REPRO Federation
    - updated “Rolling Calibration” constants
- Each of these federations is “responsible” for managing of its own set of conditions.
    - See the tables on the next pages...
- Each federation (including the “consumer” ones) has:
  - a superset of all conditions produced by each individual federation

The Federation Setup at SLAC: Distributed Conditions

Idea: *The conditions are distributed (created/updated) between 3 federations. Each federation has a copy of conditions from others.*

Technology: *Two-layered namespace* **<detector>/<condition>** *is mappeed to* **<origin>-***s.*
Inter-federations synchronization mechanisms.



The Federation Setup at SLAC: Statistics

Table: *There are more than 400 grouped into 10 detectors.*

Subsystem / Federation	IR2	OPR	REPRO/OPR
Dch	24	55	3
Pep	3	11	11
Trk	-	1	-
Svt	25	10	2
Ifr	24	1	1
Drc	38	10	2
Emc	71	26	16
Orc	53	-	-
Mat	-	3	-
Dct	51	-	-
Subtotal:	289	117	35 (shared)
Total:	**406**		

The Configuration files of public federations

- Location
  - **$BFROOT/Bdb/Revisions/<sub-dir>**
  - The files will have names similar to the ones of the data sets:
    - **2000/b1-s0-r8B-on0.V01**
    - See more details on naming from:
      - http://www.slac.stanford.edu/BFROOT/www/Physics/BaBarData/GoodRuns/dataSets.html
  - See sample configuration files at:
    - **$BFROOT/Bdb/Revisions/2000-Test/**
- When updated
  - All conditions at the end of the reprocessing
    - Is done semi-automatically
    - Expected frequency: 5..10 times per year.
  - When alignments are updated
    - Is done manually by an author of modification
    - Expected frequency: as required.
- How to use configuration files
  - A question of a policy???

Typical Use Cases

- New revisions at the end of reprocessing of a data set.
- New revisions for the IR2 conditions.
- New revisions when detector alignments or other manually updated conditions are changed.
- Delete existing revision:
  - Is technically possible. Should be avoided except very special cases.
- ???