Tutorial topics

- MySQL Cluster Overview
- MySQL Cluster Components
- Partitioning
- Internal Blocks
- General Configuration
- Transactions
- Checkpoints on disk
- Handling failure
- Disk data
- Start Types and Start Phases
- Storage Requirements
- Backups
- NDB and binlog
- Native NDB Online Backups

What this course is not?

This course doesn't cover advanced topics like:

Advanced Setup

Performance Tuning

Configuration of multi-threads

Online add nodes

NDBAPI

noSQL access

Geographic Replication

MySQL Cluster Overview

MySQL Cluster Overview

High Availability Storage Engine for MySQL

Provides:

- High Availability
- Clustering
- Horizontal Scalability
- Shared Nothing Architecture
- SQL and noSQL access

High Availability and Clustering

High Availability:

- survives routine failures or maintenance
  - hardware or network failure, software upgrade, etc
- provides continuous services
- no interruption of services
- 99.999% uptime (5.26 mins downtime/year)

Clustering:

- parallel processing
- increase availability

Scalability

Ability to handle increased number of requests

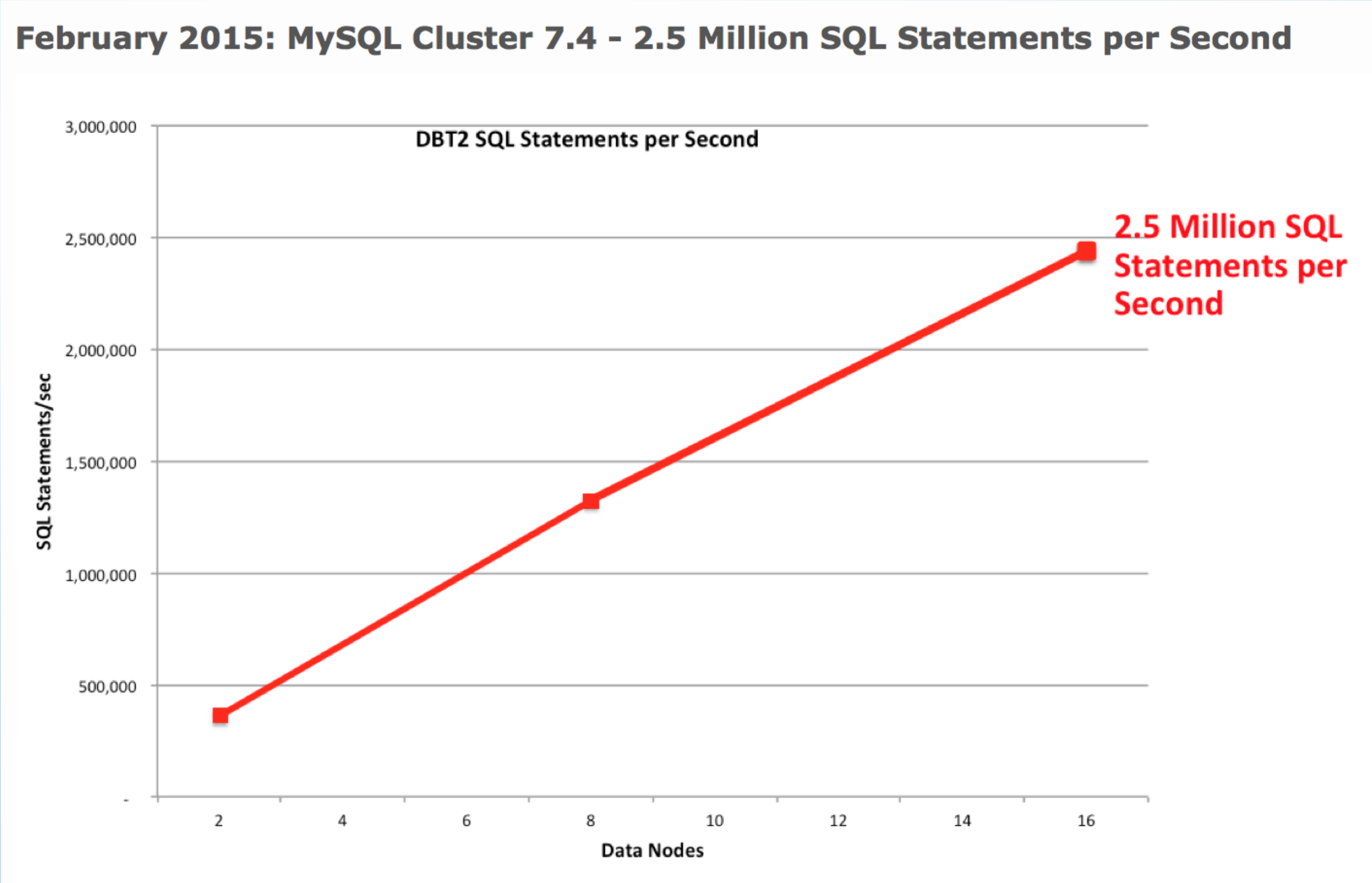
Vertical:

- replace HW with more powerful one

Horizontal:

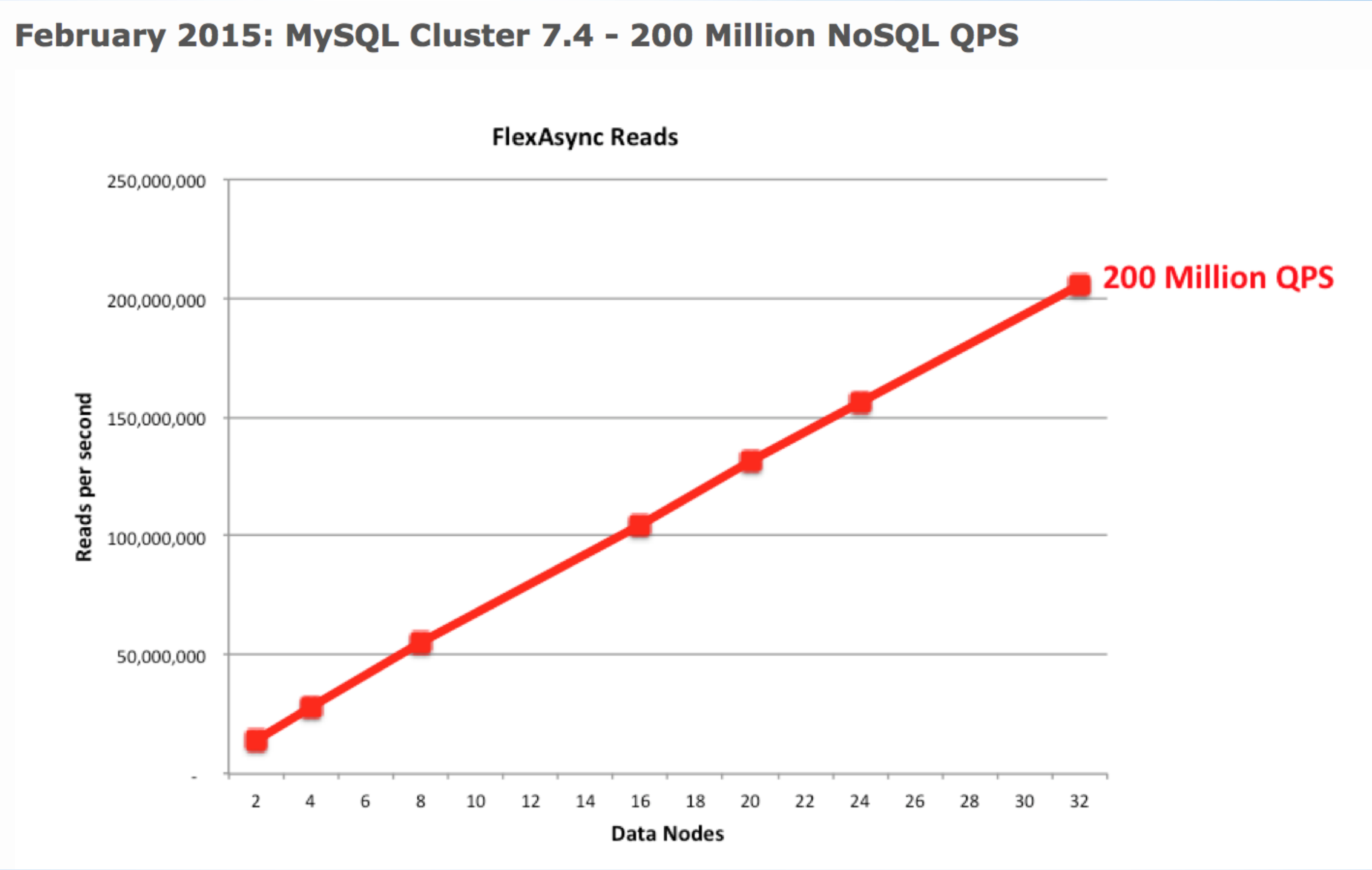
- add component to the system

Scalability



https://www.mysql.com/why-mysql/benchmarks/mysql-cluster/

Scalability



https://www.mysql.com/why-mysql/benchmarks/mysql-cluster/

Shared Nothing Architecture

Characterized by:

- No single point of failure
- Redundancy
- Automatic Failover

MySQL Cluster Summary

Shared Nothing Architecture

No Single Point Of Failure (SPOF)

Synchronous replication between nodes

Automatic failover with no data loss

ACID transaction

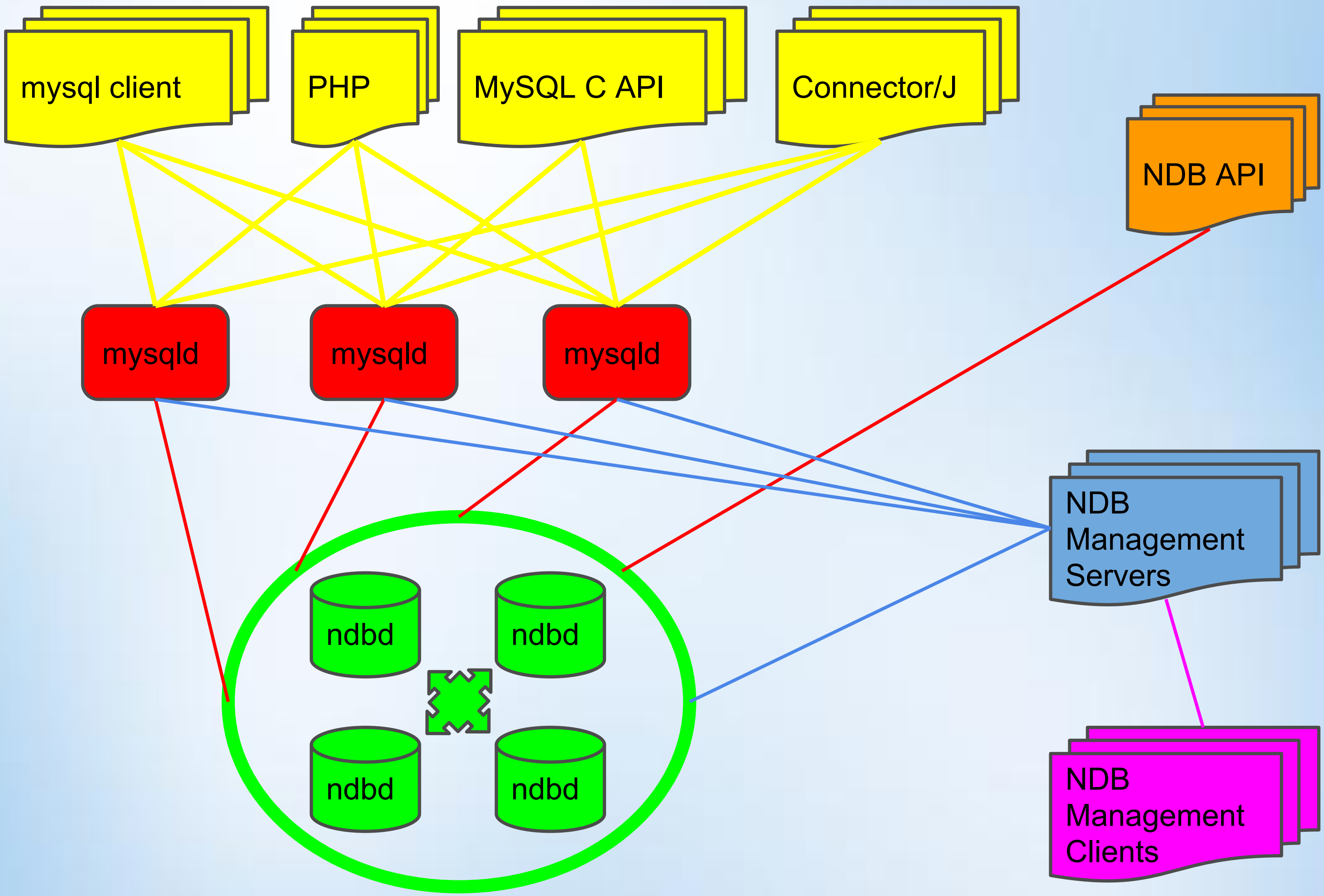
Durability with NumberOfReplica > 1

READ COMMITTED transaction isolation level

Row level locking

MySQL Cluster Components

Components Overview



Cluster Nodes

3 Node Types are present in a MySQL Cluster:

- Management Node
- Data Node
- Access Node

Management Node

ndb_mgmd process

Administrative tasks:

- configure the Cluster
- start / stop other nodes
- run backup
- monitoring
- coordinator

Data Node

ndbd process (or **ndbmtd**)

Stores cluster data

- mainly in memory
- also on disk (limitations apply)

Coordinate transactions

Data Node (ndbmtd)

What is ndbmtd ?

Multi-threaded equivalent of ndbd

File system compatible with ndbd

By default run in single thread mode

Configured via MaxNoOfExecutionThreads or ThreadConfig

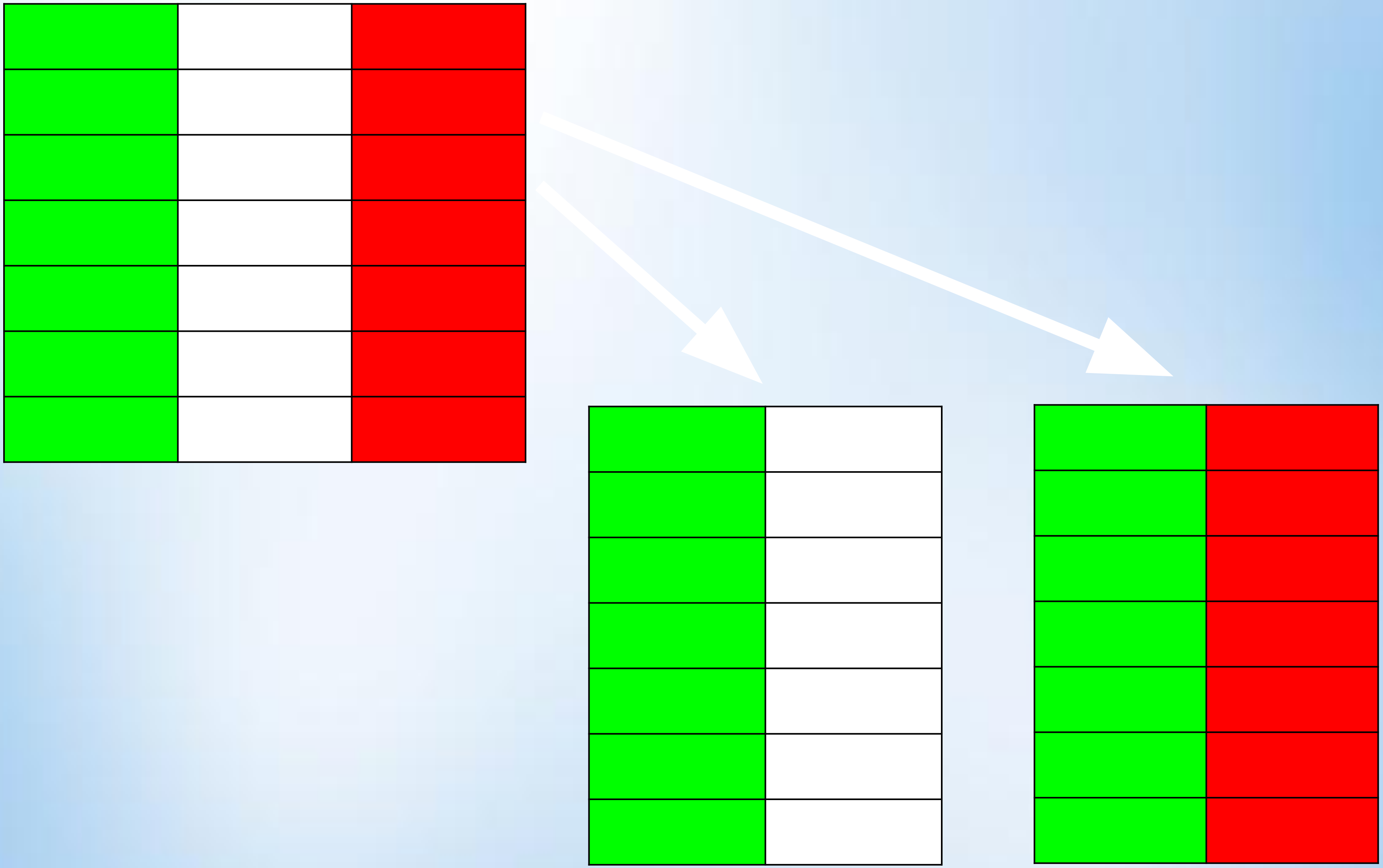
API Node

Process able to access data stored in Data Nodes:

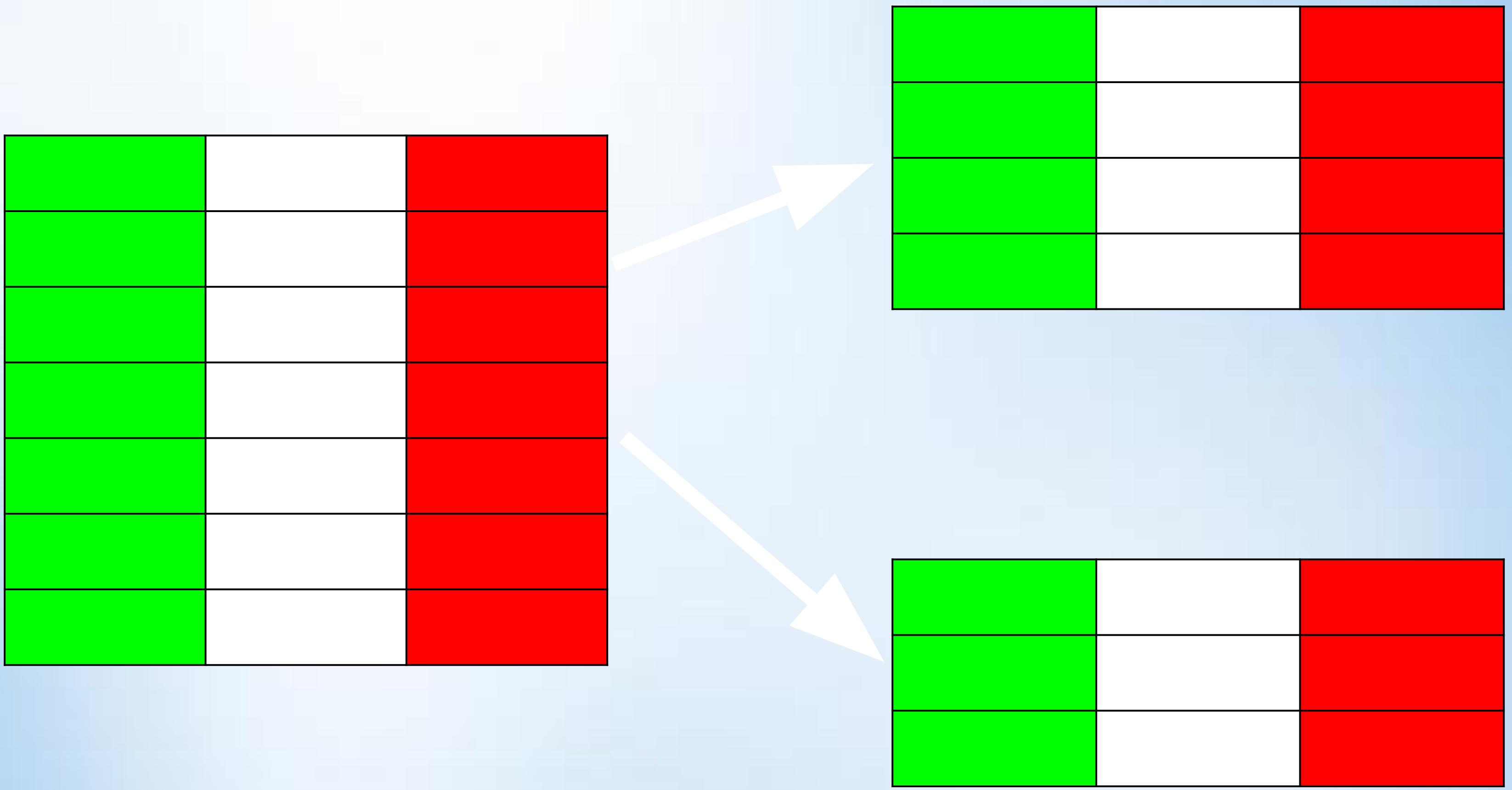
- **mysqld** process with NDBCLUSTER support
- application that connects to the cluster through NDB API (without mysqld)

Partitioning

Vertical Partitioning



Horizontal Partitioning



Partitions example (1/2)

Table

5681	Alfred	30	2010-11-10
8675	Lisa	34	1971-01-12
5645	Mark	21	1982-01-02
0965	Josh	36	2009-06-33
5843	Allan	22	2007-04-12
1297	Albert	40	2000-12-20
1875	Anne	34	1984-11-22
9346	Mary	22	1983-08-05
8234	Isaac	20	1977-07-06
8839	Leonardo	28	1999-11-08
7760	Donna	37	1997-03-04
3301	Ted	33	2005-05-23

Partitions example (2/2)

Table

5681	Alfred	30	2010-11-10	
8675	Lisa	34	1971-01-12	P1
5645	Mark	21	1982-01-02	
0965	Josh	36	2009-06-33	
5843	Allan	22	2007-04-12	P2
1297	Albert	40	2000-12-20	
1875	Anne	34	1984-11-22	
9346	Mary	22	1983-08-05	P3
8234	Isaac	20	1977-07-06	
8839	Leonardo	28	1999-11-08	
7760	Donna	37	1997-03-04	P4
3301	Ted	33	2005-05-23	

Partitioning in MySQL Cluster

MySQL Cluster natively supports Partitioning

Distribute portions of individual tables in different locations

PARTITION BY (LINEAR) KEY is the only partitioning type supported

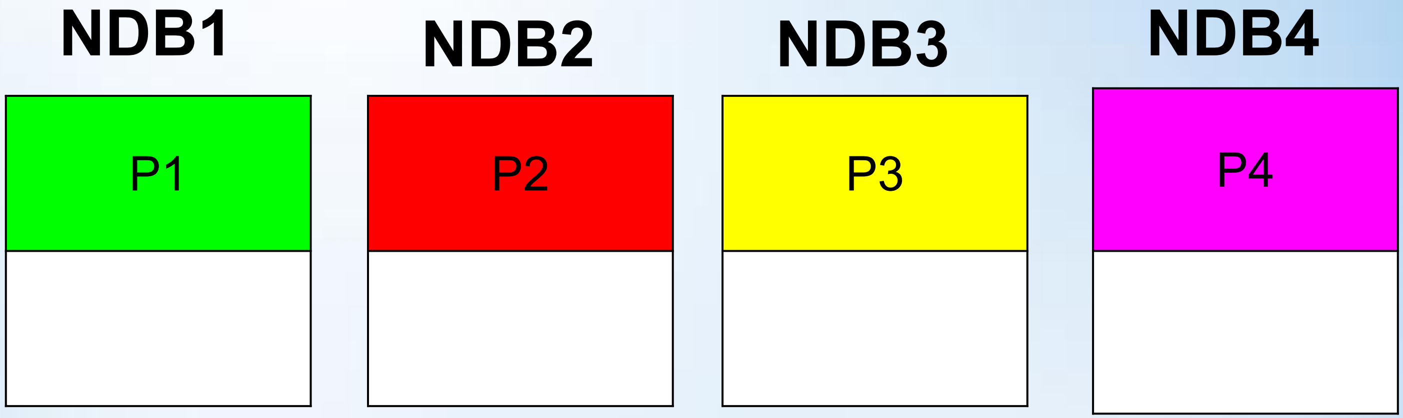
Partitions, Node Groups & Replicas

Partition:
portion of the data stored in the cluster

Node Groups:
set of data nodes that stores a set of partitions

Replicas:
copies of a cluster partition

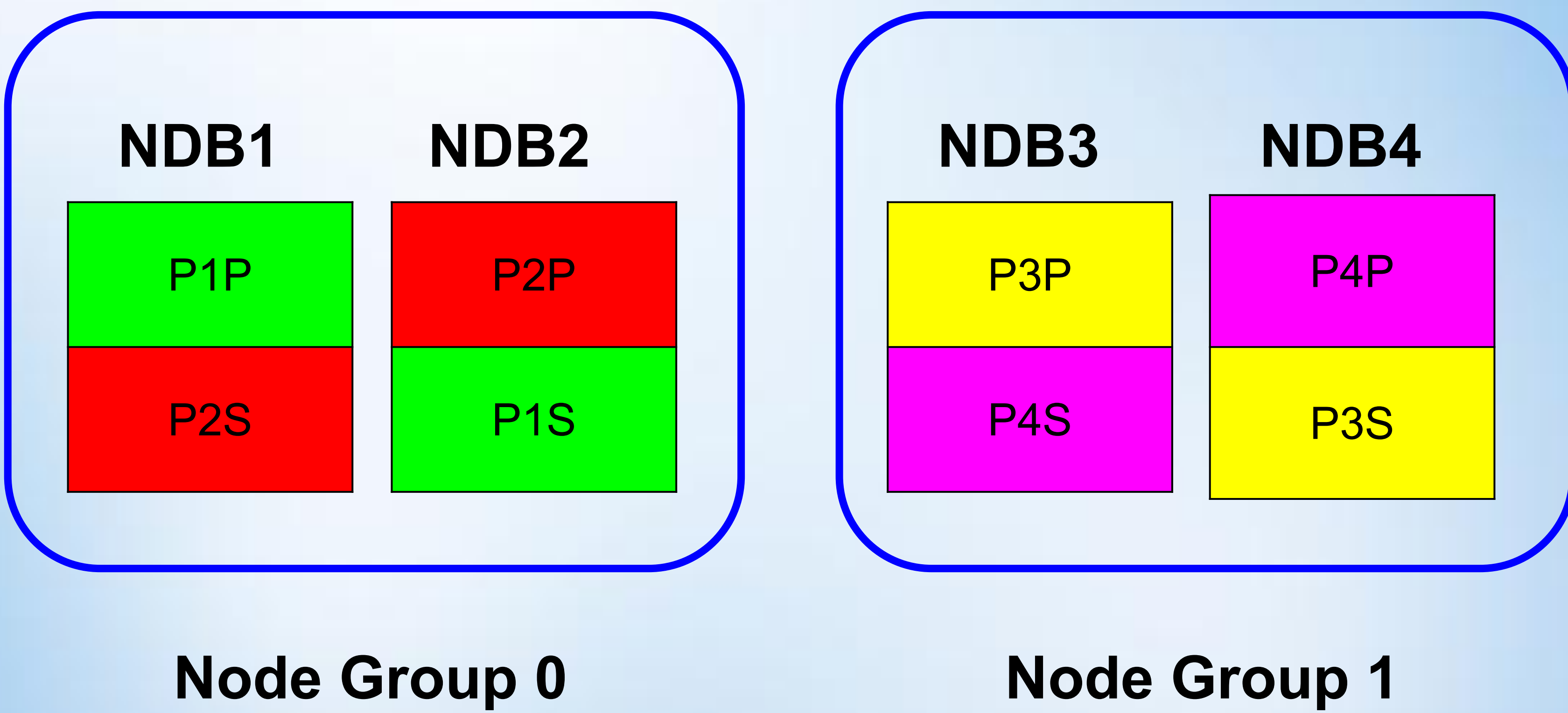
Partitions in NDB



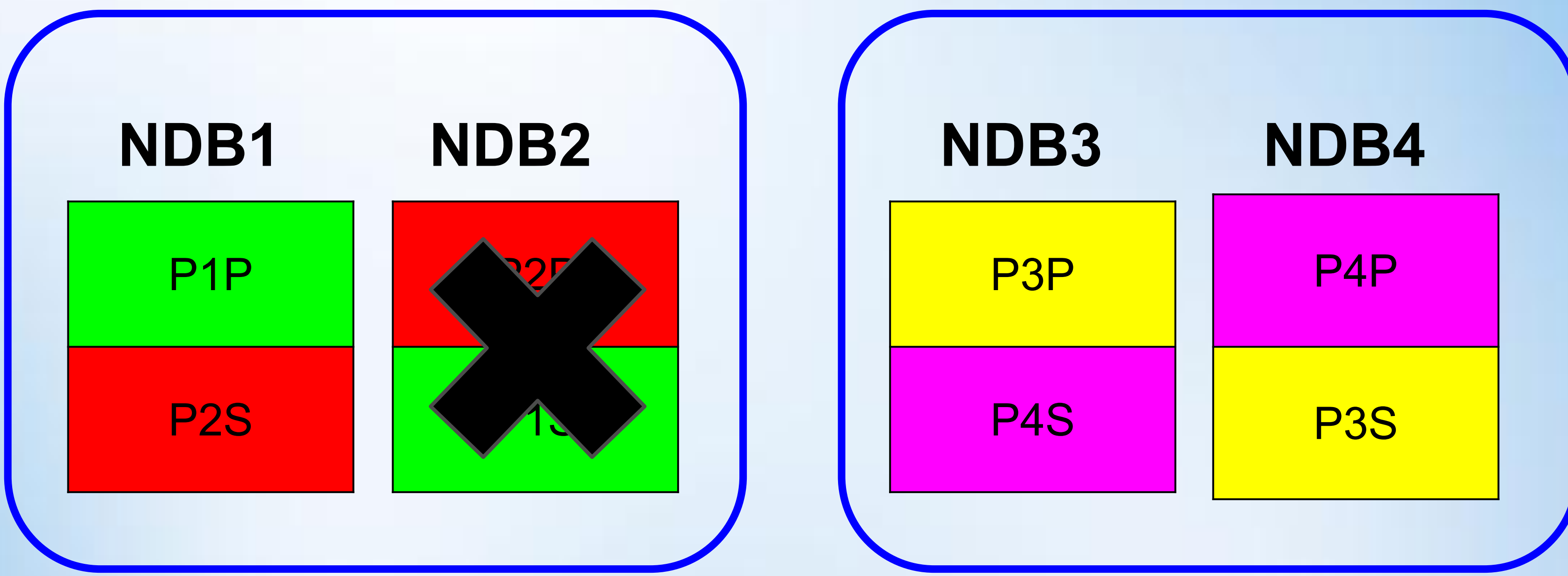
Partitions and Replicas

NDB1	NDB2	NDB3	NDB4
P1P	P2P	P3P	P4P
P2S	P1S	P4S	P3S

Node Groups


NDB1
NDB2
P1P
P2P
P2S
P1S
NDB3
NDB4
P3P
P4P
P4S
P3S
Node Group 0
Node Group 1

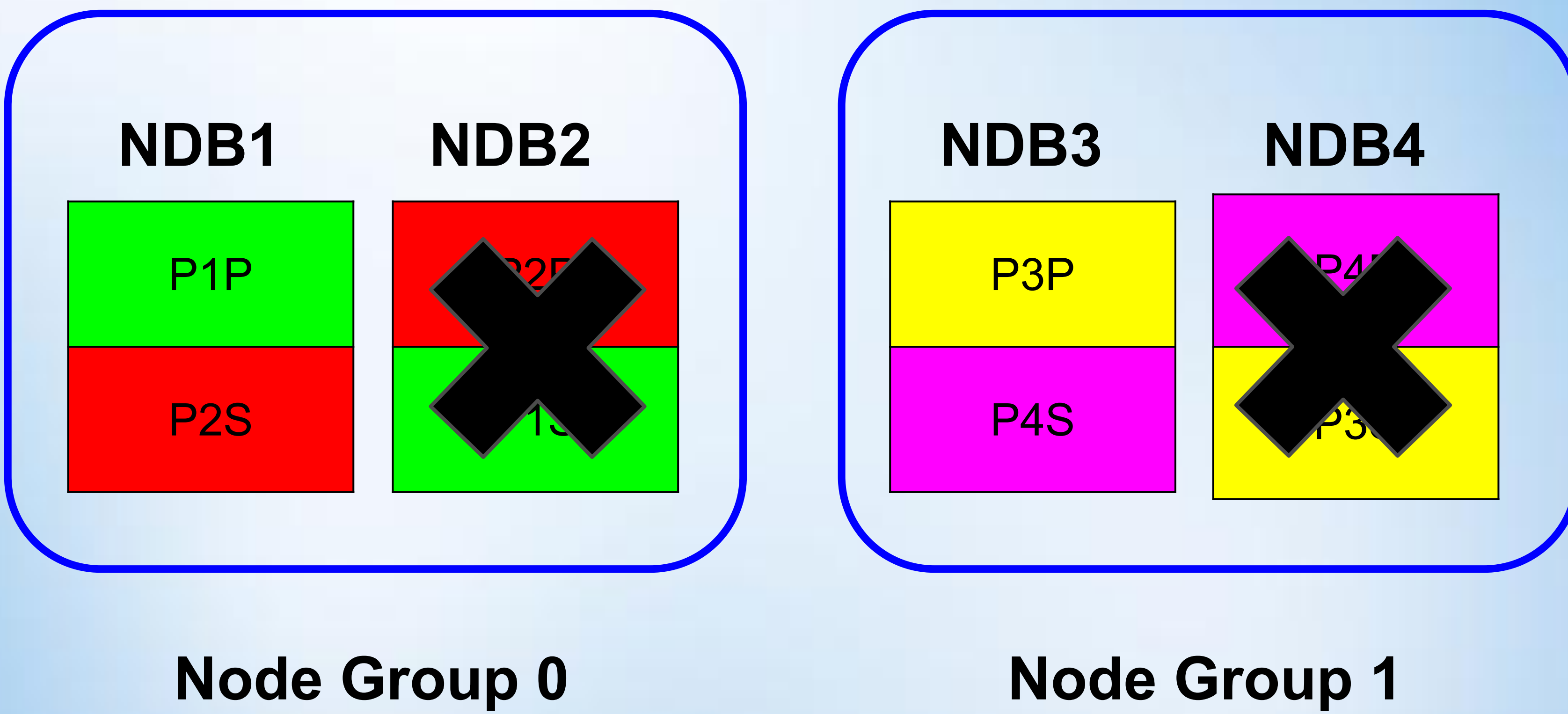
Node Groups



Node Group 0

Node Group 1

Node Groups



Internal Blocks

Blocks in Data Node (1/2)

A data node is internally structured in several blocks/code with different functions:

- Transaction Coordinator (TC) :
  - coordinates transactions
  - starts Two Phase Commit
  - distributes requests to LQH
- Local Query Handler (LQH)
  - allocates local operation records
  - manages the Redo Log

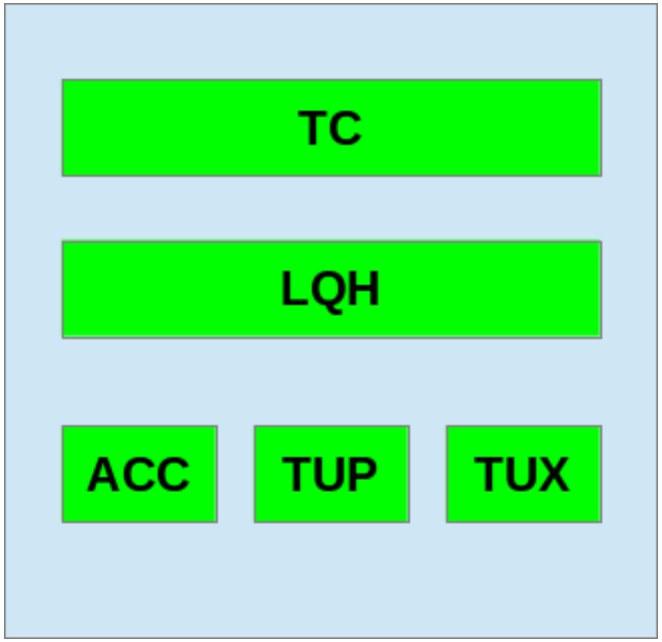
Blocks in Data Node (2/2)

- ACC
  - stores hash indexes
  - manages records locking
- TUP
  - stores row data
- TUX
  - stores ordered indexes
- BACKUP , CMVMI , DICT , DIH , SUMA , etc

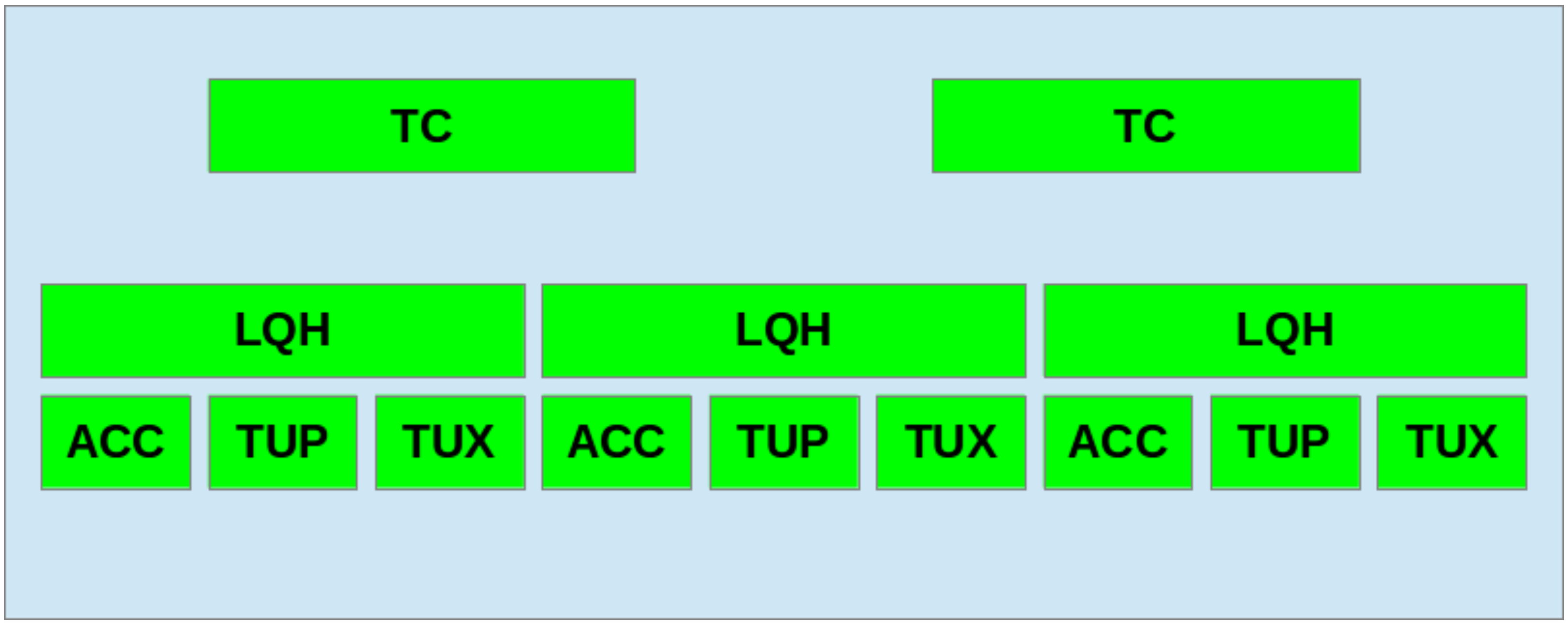
Ref:

http://dev.mysql.com/doc/ndbapi/en/ndb-internals-ndb-protocol.html

Blocks on ndbd (simplified)

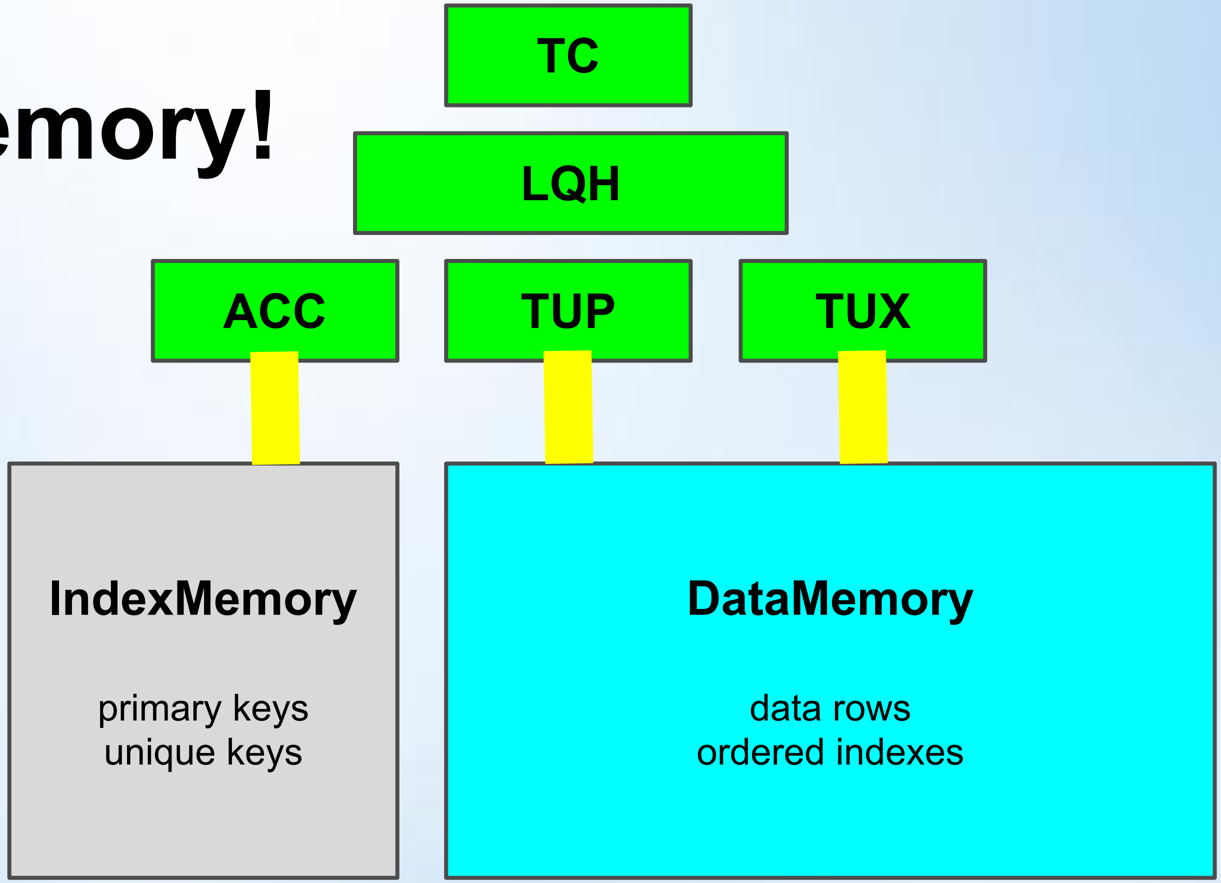


Blocks on ndbmtd (simplified)



Where data is stored?

In memory!



General Configuration

Global parameters

NoOfReplicas (mandatory)

Defines the number of replicas for each partition/table

DataDir:

Location for log/trace files and pid file

FileSystemPath:

location for of all files related to MySQL Cluster

Global boolean parameters

LockPagesInMainMemory (0) :

Lock all memory in RAM (no swap)

StopOnError (1) :

Automatic restart of data node in case of failure

IndexMemory and DataMemory

IndexMemory: defines the amount of memory for hashes indexes (PK and UNIQUE)

DataMemory: defines the amount of memory for:

- data
- ordered indexes
- UNDO information

Metadata Objects

MaxNoOfTables

MaxNoOfOrderedIndexes

MaxNoOfUniqueHashIndexes

Transactions

Transactions Overview

Two Phase Commit

ACID

- Durability with multiple copies
- Committed on memory

READ_COMMITTED

Transaction implementation

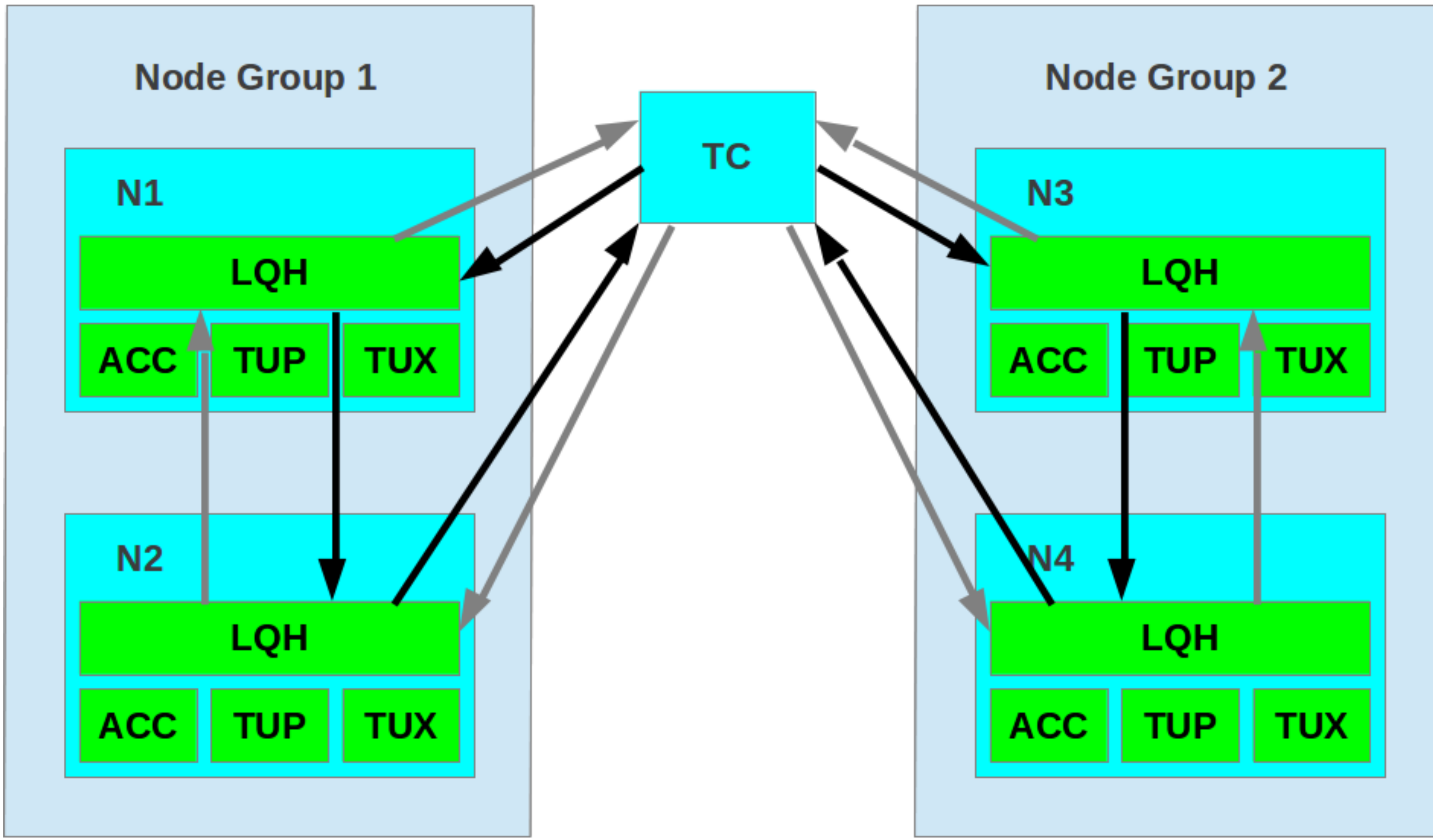
- The API node contacts a TC in one of the Data Node (round-robin fashion, or Distribution Awareness)
- The TC starts the transaction and the Two Phase Commit (2PC) protocol
- The TC contacts the LQH of the Data Nodes involved in the transaction
- LQH handles row level locking:
  - Exclusive lock
  - Shared lock
  - No lock

Two Phase Commit (1/2)

Phase 1 , Prepare :
Node groups get their information updated

Phase 2 , Commit :
the change is committed

Two Phase Commit (2/2)



Rows operations

Primary key operation (read and write)

Unique index operation (read and write)

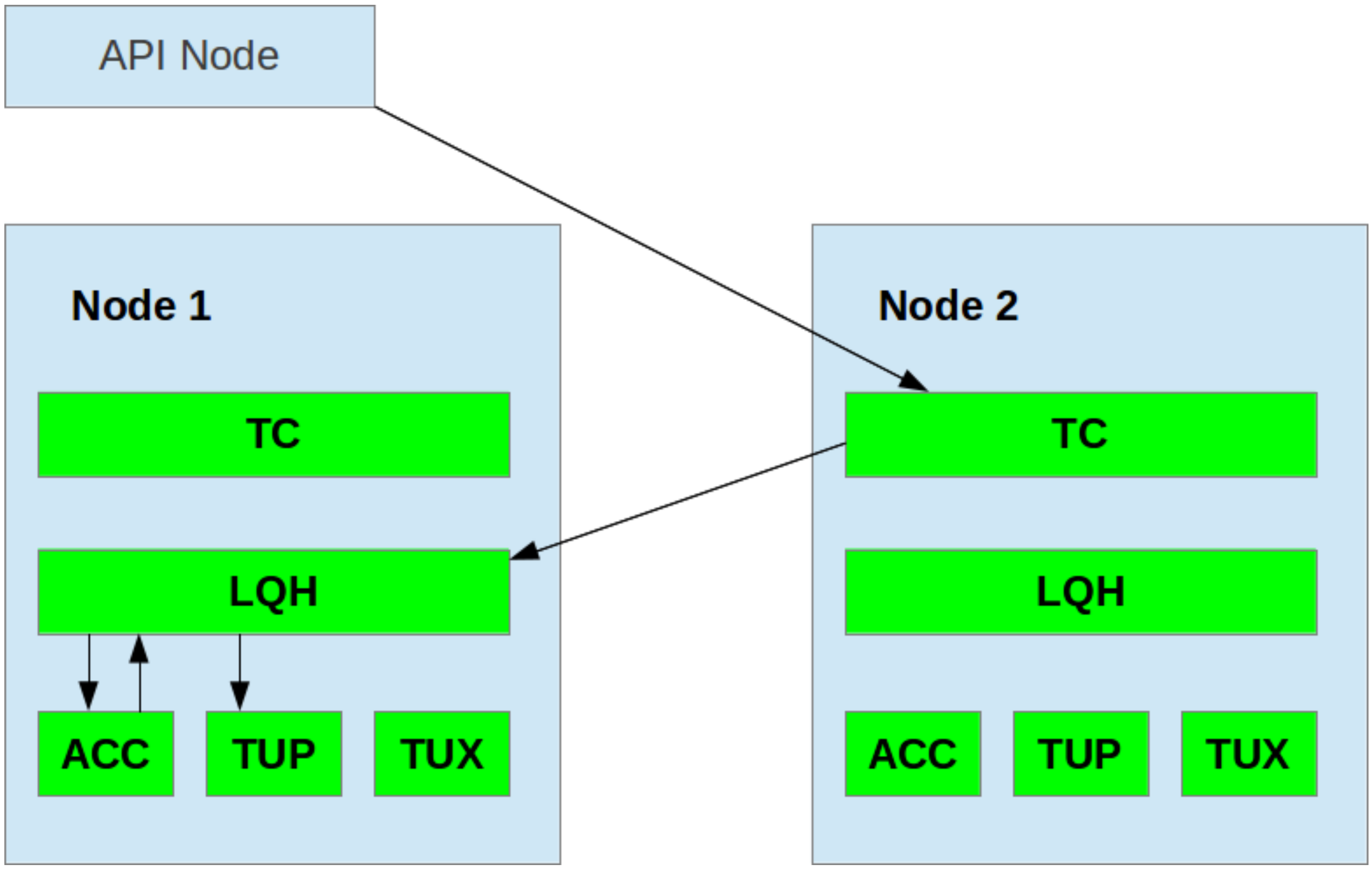
Ordered index scans (read only)

Full-table scans (read only)

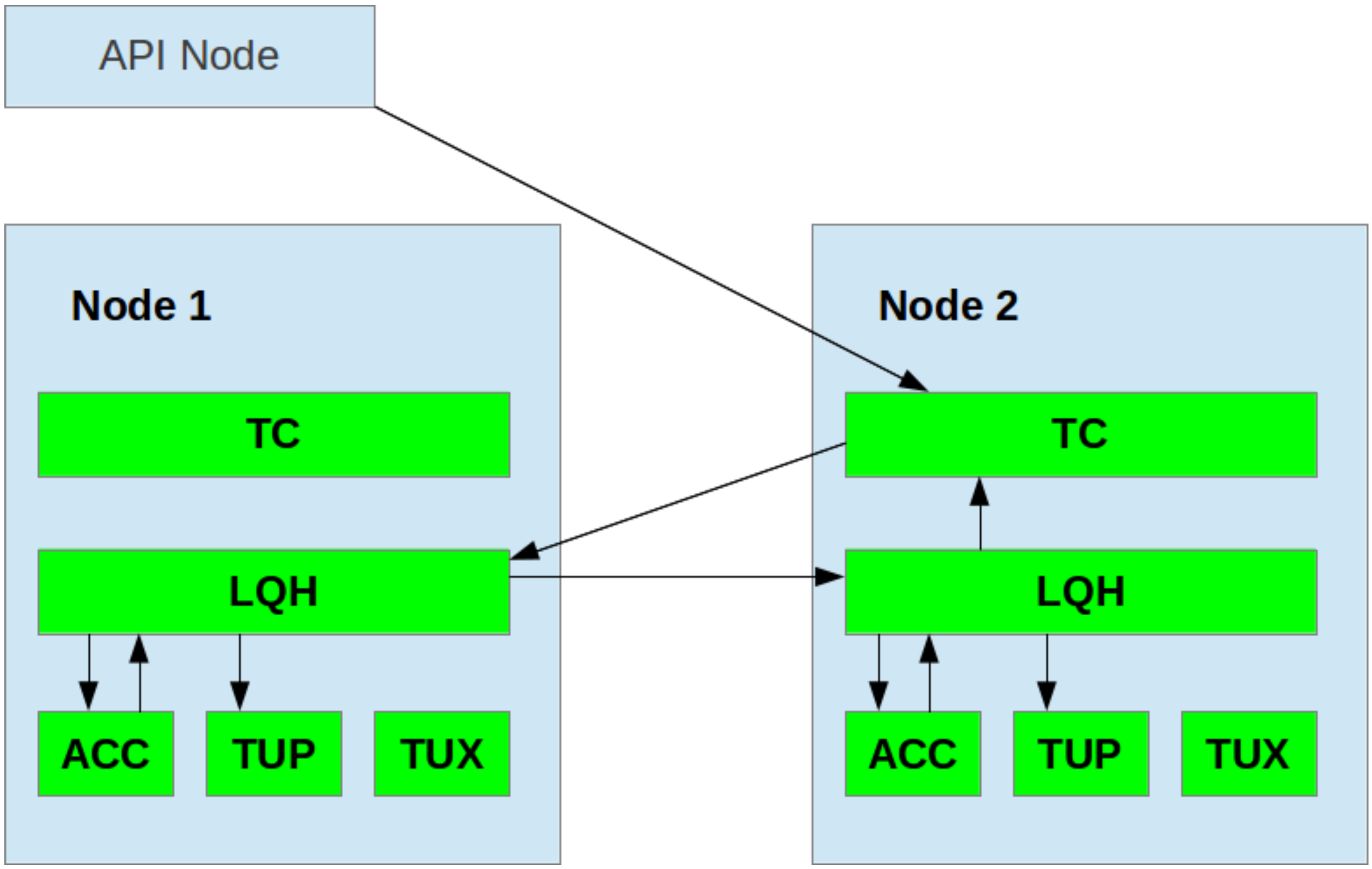
Primary Key Write (1/4)

- API node connects to TC
- TC connects to LQH of the data node with the primary fragment
- LQH queries ACC to get the row id
- LQH performs the operation on TUP
- LQH connects to the LQH of the data node with the secondary fragment
- LQH on secondary performs the same operation
- LQH on secondary informs TC that the prepare phase is completed
- TC starts the commit phase

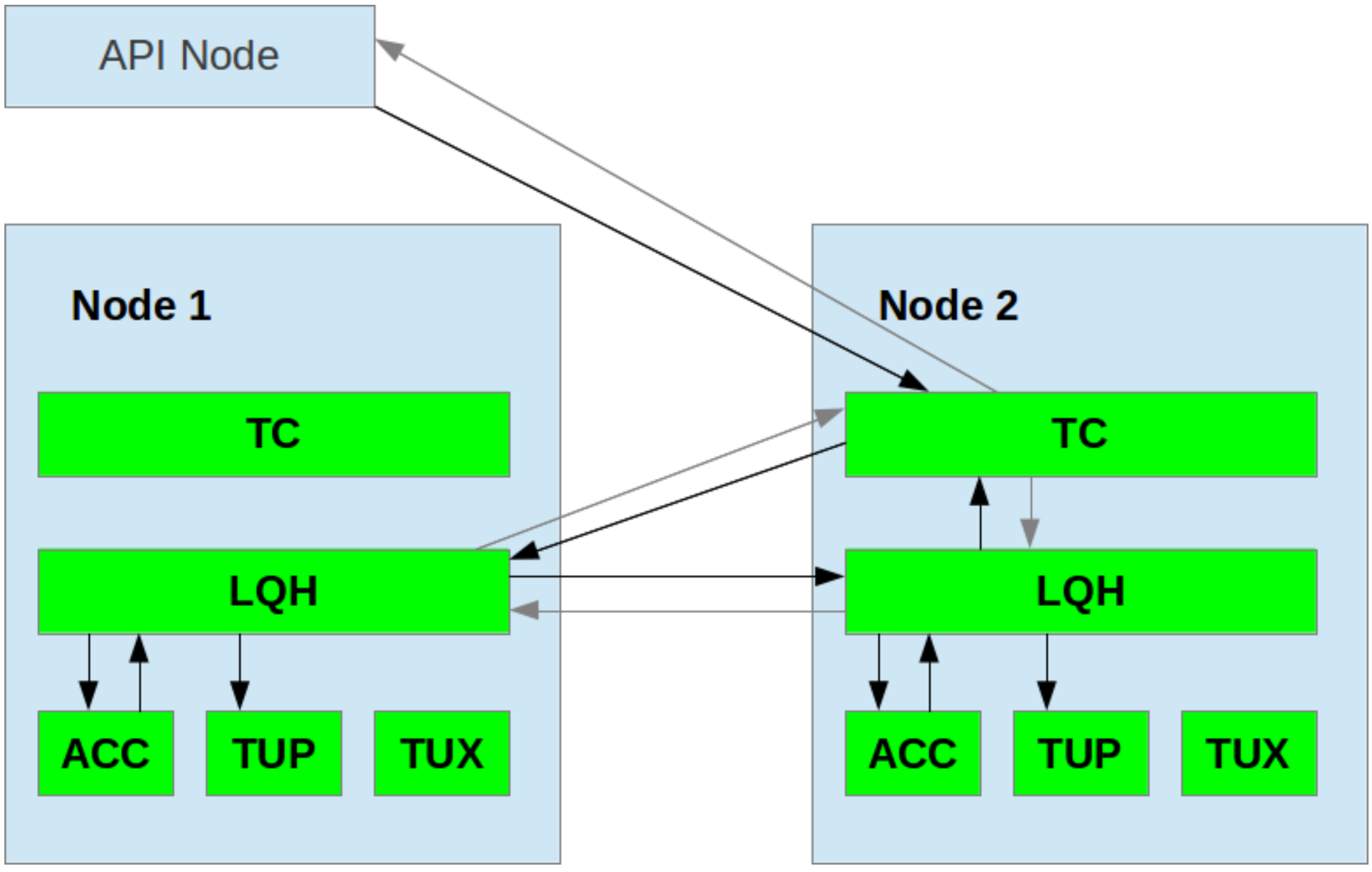
Primary Key Write (2/4)



Primary Key Write (3/4)

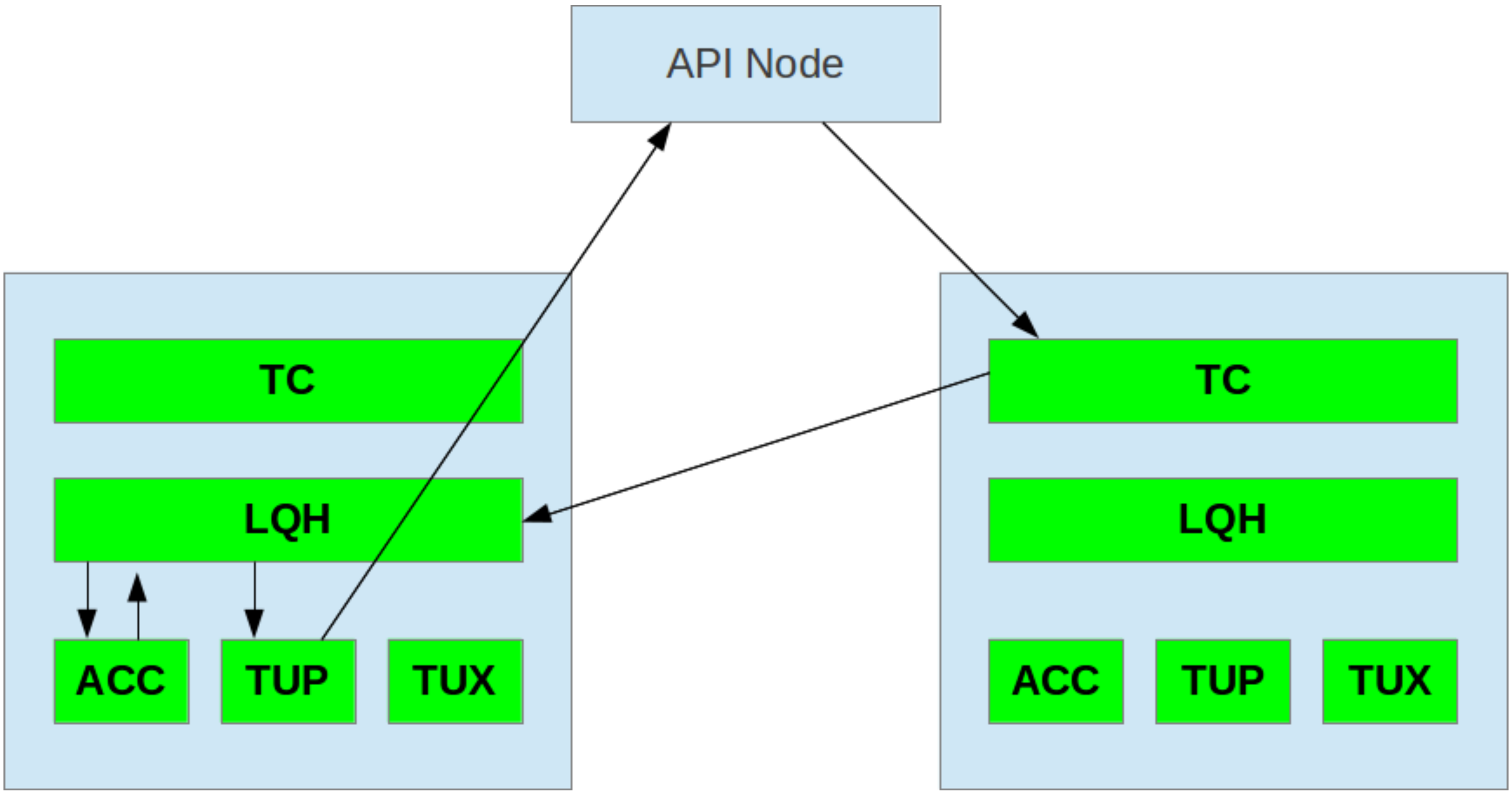


Primary Key Write (4/4)



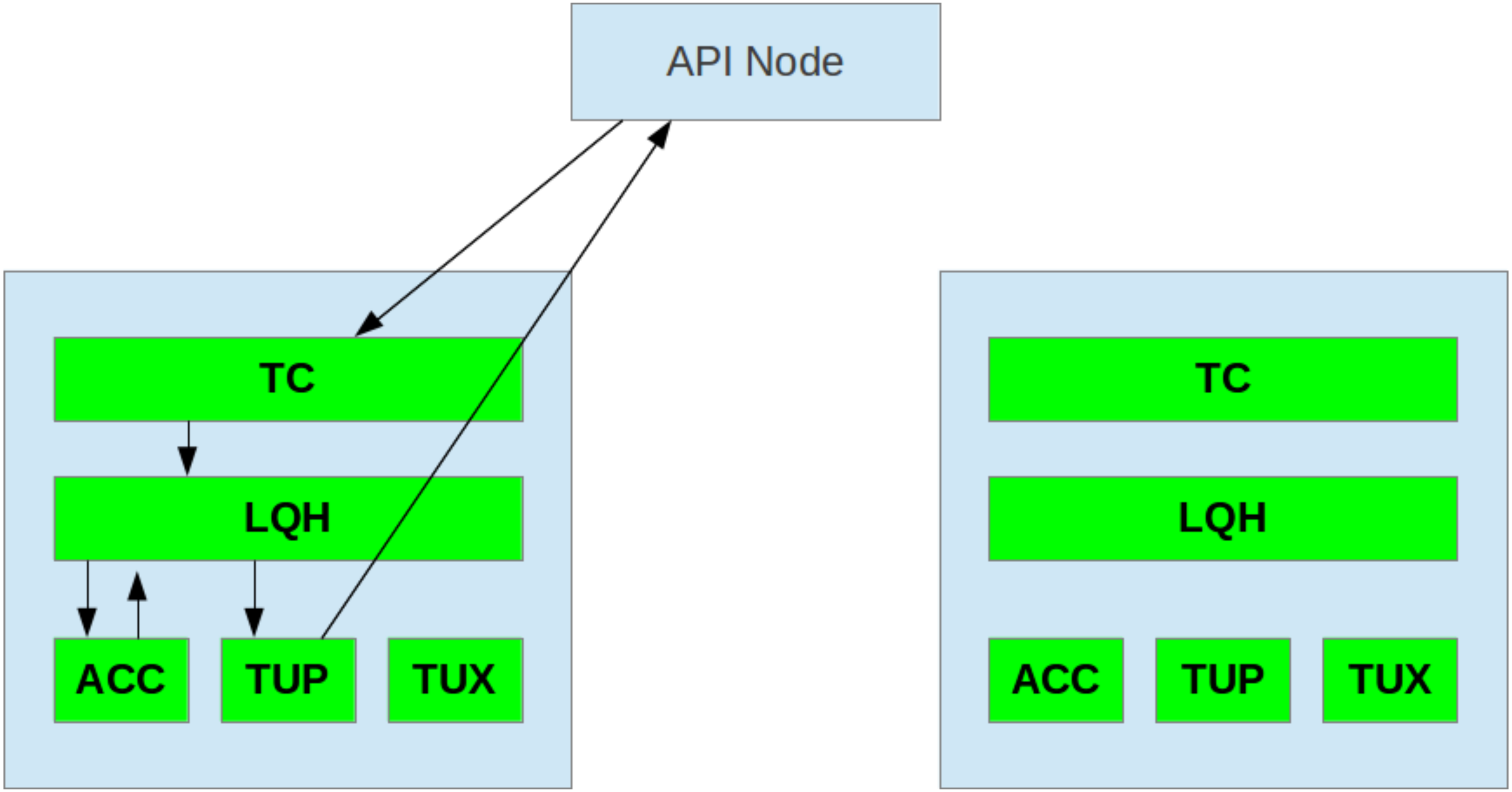
Primary Key Read

Without distribution awareness



Primary Key Read

With distribution awareness



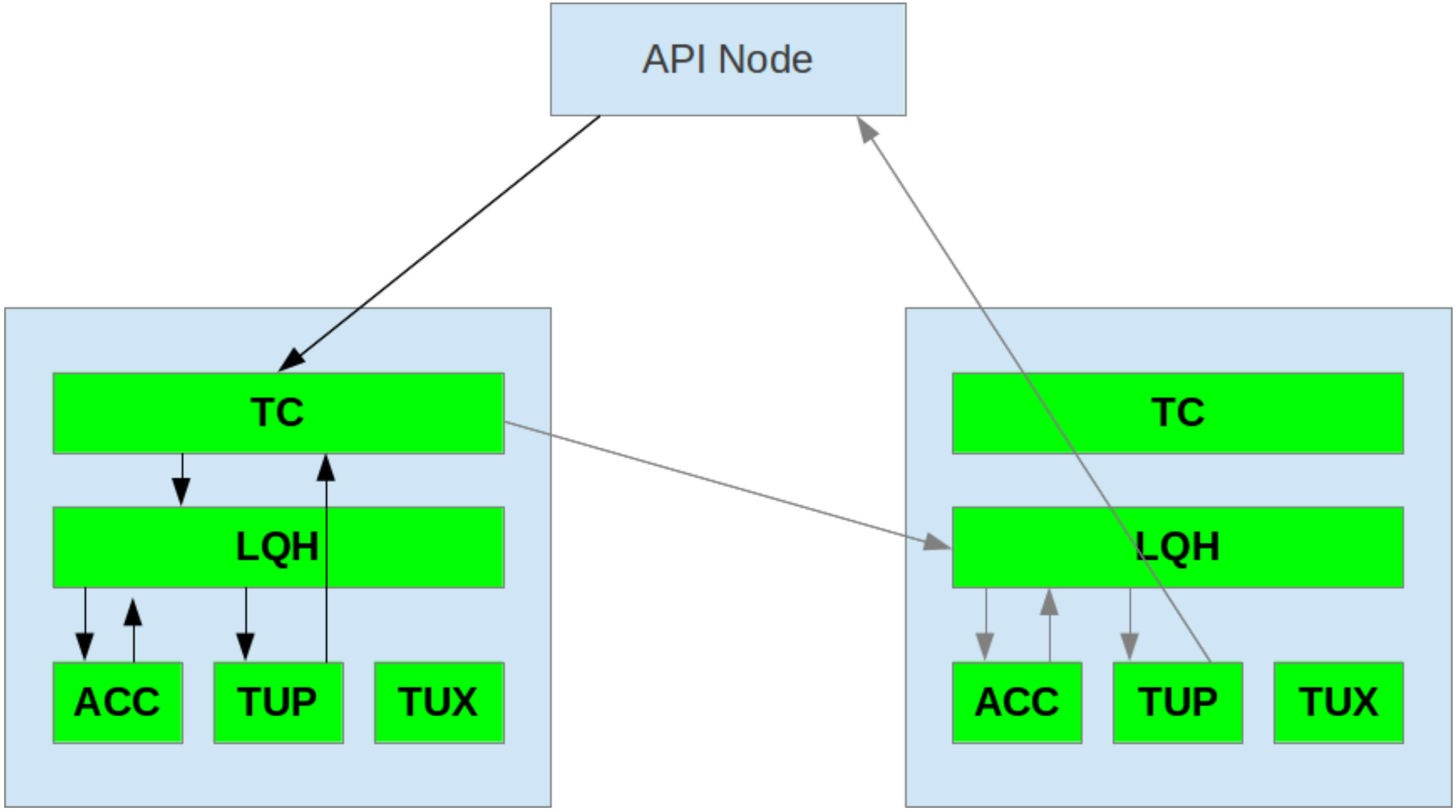
Unique Index Read (1/2)

Unique Index is internally implemented as a hidden table with two columns where the PK is the Unique Key itself and the second column is the PK of the main table.

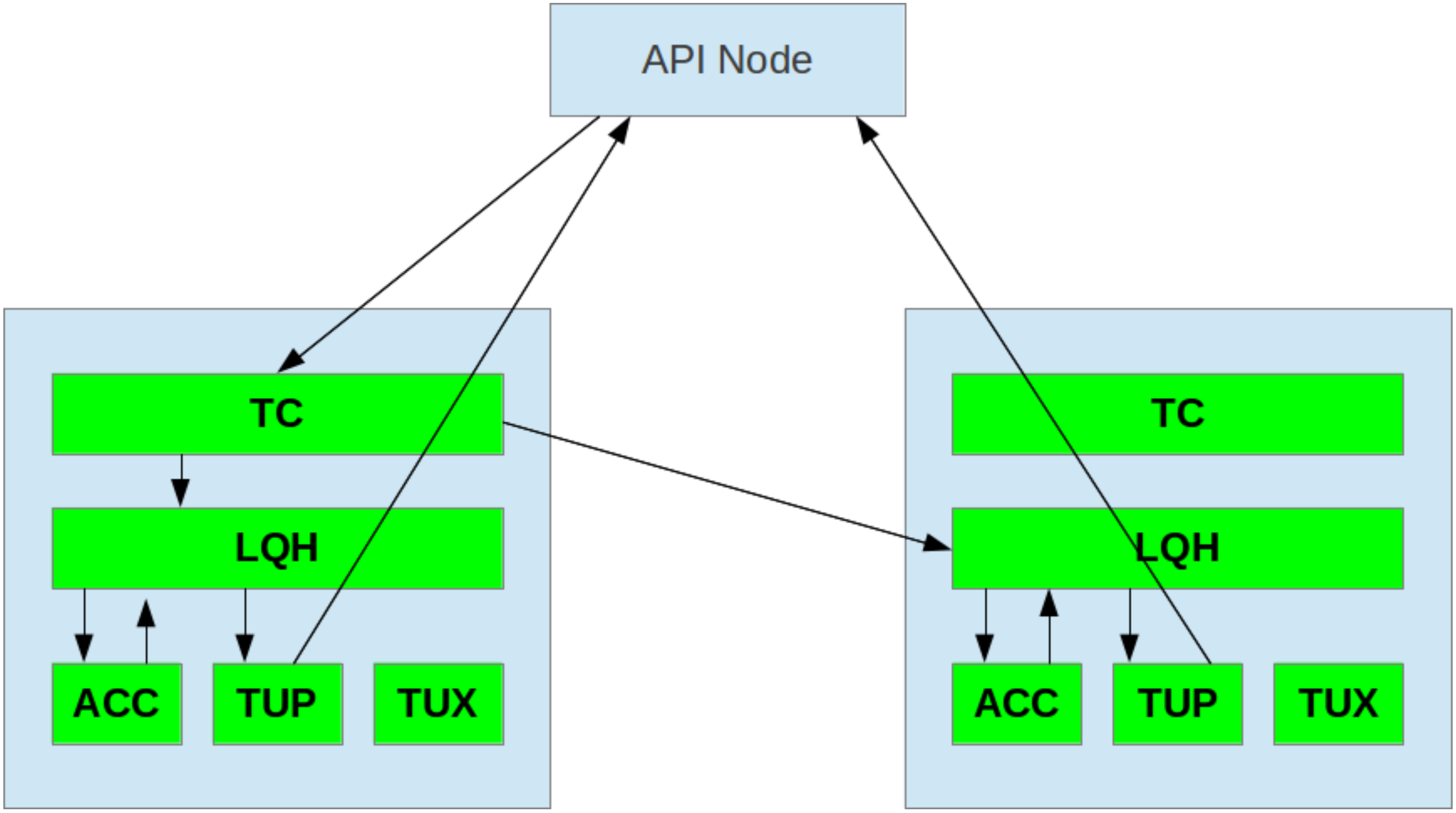
To read a Unique index:

- A PK read is performed on hidden table to get the value of the PK of main table
- A PK read is performed on main table

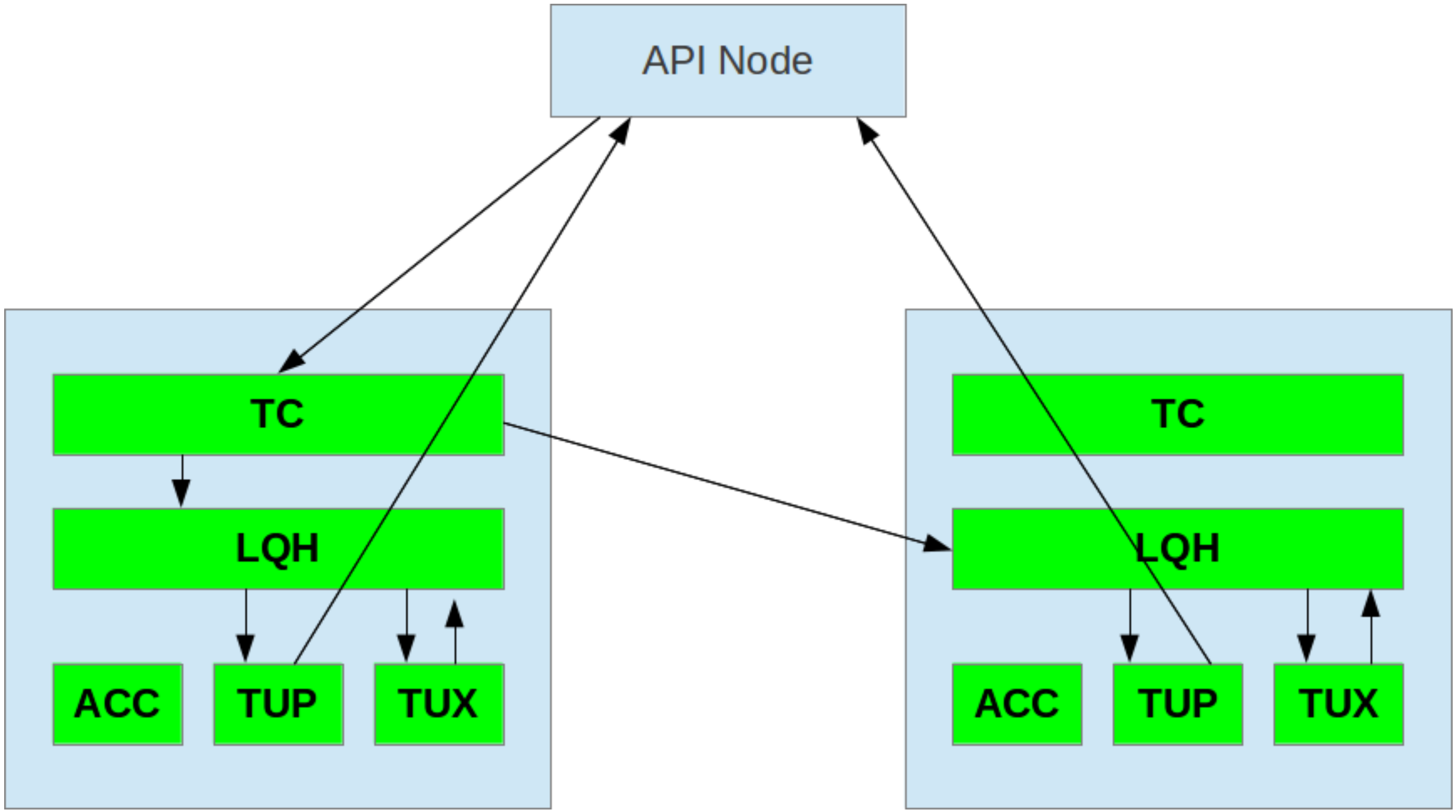
Unique Index Read (2/2)



Full Table Scan



Ordered Index Scan



Transaction Parameters (1/3)

MaxNoOfConcurrentTransactions: number of simultaneous transactions that can run in a node

(maximum number of tables accessed in any single transaction + 1)

* number of cluster SQL nodes

Transaction Parameters (2/3)

MaxNoOfConcurrentOperations:

Total number of records that at any time can be updated or locked in data node, and UNDO.

Beside the data node, TC needs information about all the records involved in a transaction

MaxNoOfLocalOperations: for not too large transactions, it defaults to MaxNoOfConcurrentOperationsx1.1

Transaction Parameters (3/3)

MaxDMLOperationsPerTransaction

Limits the number of DML operations

Count is in # of records, not # of statements

TransactionInactiveTimeout

Idle transactions are rolled back

TransactionDeadlockDetectionTimeout

Time based deadlock and lock wait detection

Also possible that the data node is dead or overloaded

Checkpoints on disk

Checkpoints

MySQL Cluster is a in-memory storage engine:

- data is only in memory? **WRONG!**
- data is regularly saved on disk

When a data node writes on disk is performing a checkpoint

Exception:

MySQL Cluster can also run in DiskLess mode, with "reduced durability" and unable to run Native NDB Backup

Checkpoint types

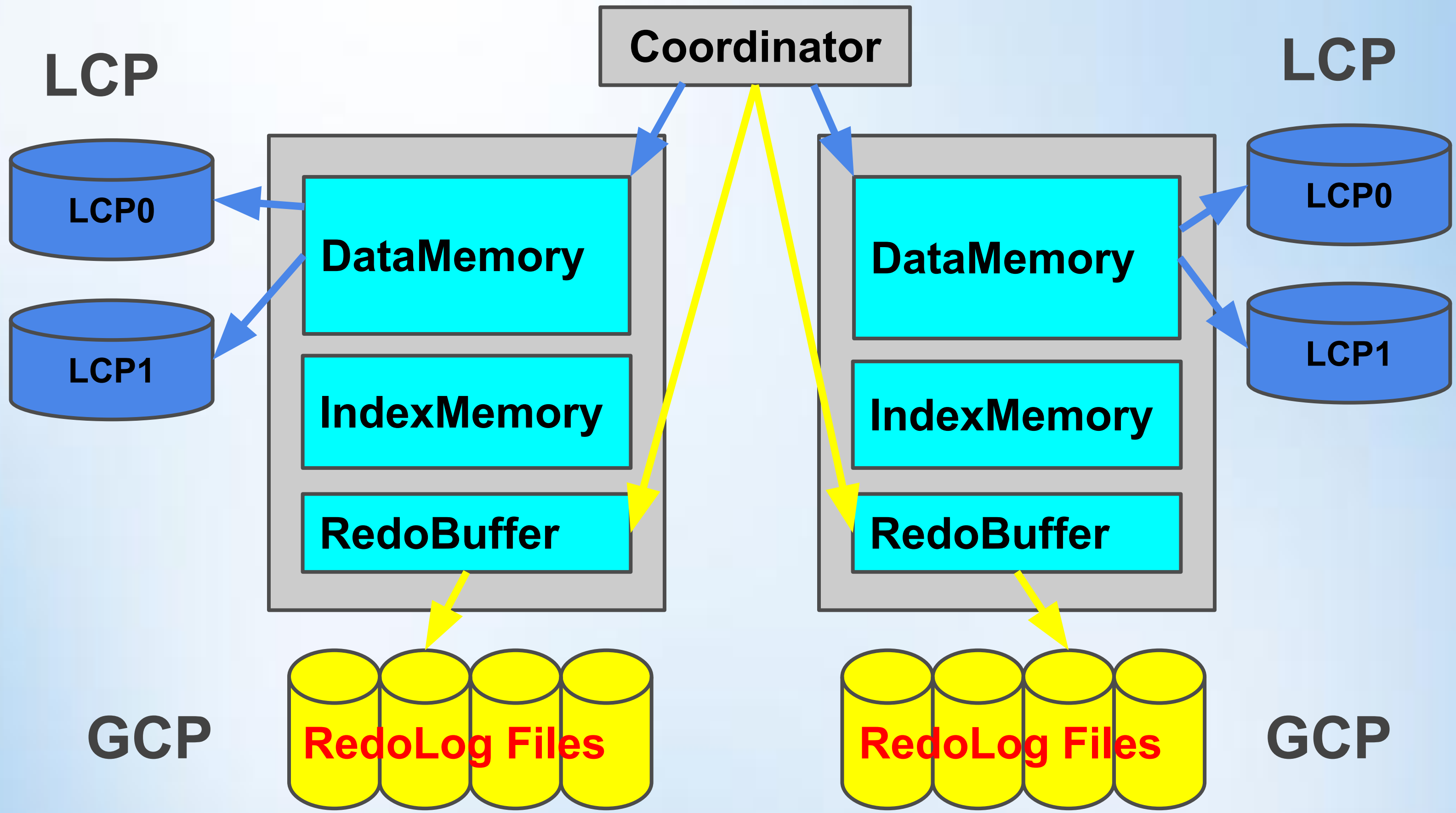
Local Checkpoint (LCP):

- specific to a single node;
- all data in memory is saved to disk
- can take minutes or hours

Global Checkpoint (GCP):

- transactions across nodes are synchronized
- redo log flushed on disk

Memory and disk checkpoints



Undo and Redo Parameters

UndoIndexBuffer and UndoDataBuffer

Buffers used during local checkpoint to perform consistent snapshot without blocking writes.

RedoBuffer

In memory buffer for REDO log (on disk)

Checkpointing parameters

TimeBetweenLocalCheckpoints

Doesn't define a time interval, but the amount of writes operations (as a base-2 logarithm of 4-byte words) .
Examples:

20 (default) means 4 x 2^{20} = 4MB

24 means 4 x 2^{24} = 64MB

TimeBetweenGlobalCheckpoints

Defines how often commits are flushed to REDO logs

Fragment Log Files

NoOfFragmentLogFiles (default 16)

FragmentLogFileSize (default 16 MB)

REDO logs are organized in a ring, where head and tail never meet. If checkpointing is slow, updating transactions are aborted. Total REDO logs size is defined as:

NoOfFragmentLogFiles set of 4 FragmentLogFileSize

Default : 16 x 4 x 16MB = 1024MB

InitFragmentLogFiles : SPARSE or FULL

Checkpoint write control

ODirect: (off) O_DIRECT for LCP and GCP

CompressedLCP: (off) equivalent of *gzip --fast*

DiskCheckpointSpeed: speed of local checkpoints to disk (default 10MB/s)

DiskCheckpointSpeedInRestart : during restart

Diskless mode

Diskless :

It is possible to configure the whole Cluster to run without using the disk.

When Diskless is enabled (disabled by default):

- LCP and GCP are not performed;
- NDB Native Backup is not possible;
- a complete cluster failure (or shutdown) = data lost

Handling Failure

Data node failure (1/2)

Failures happen!

HW failure, network issue, OS crash, SW bug, …

MySQL Cluster is constantly monitoring itself

Each data node periodically checks other data nodes via heartbeat

Each data node periodically checks API nodes via heartbeat

Data node failure (2/2)

If a node fails to respond to heartbeats, the data node that detects the failure communicate it to other data nodes

The data nodes stop using the failed node and switch to the secondary node for the same fragment

Timeouts Parameters

TimeBetweenWatchDogCheck

The watchdog thread checks if the main thread got stuck

HeartbeatIntervalDbDb

Heartbeat interval between data nodes

HeartbeatIntervalDbApi

Heartbeat interval between data nodes and API nodes

Disk Data

Disk Data

Available since MySQL 5.1.6

Store non-indexed columns on disk

Doesn't support variable sized storage

On disk : tablespaces and undo logs

In memory : page buffer (cache)

Disk Data Objects

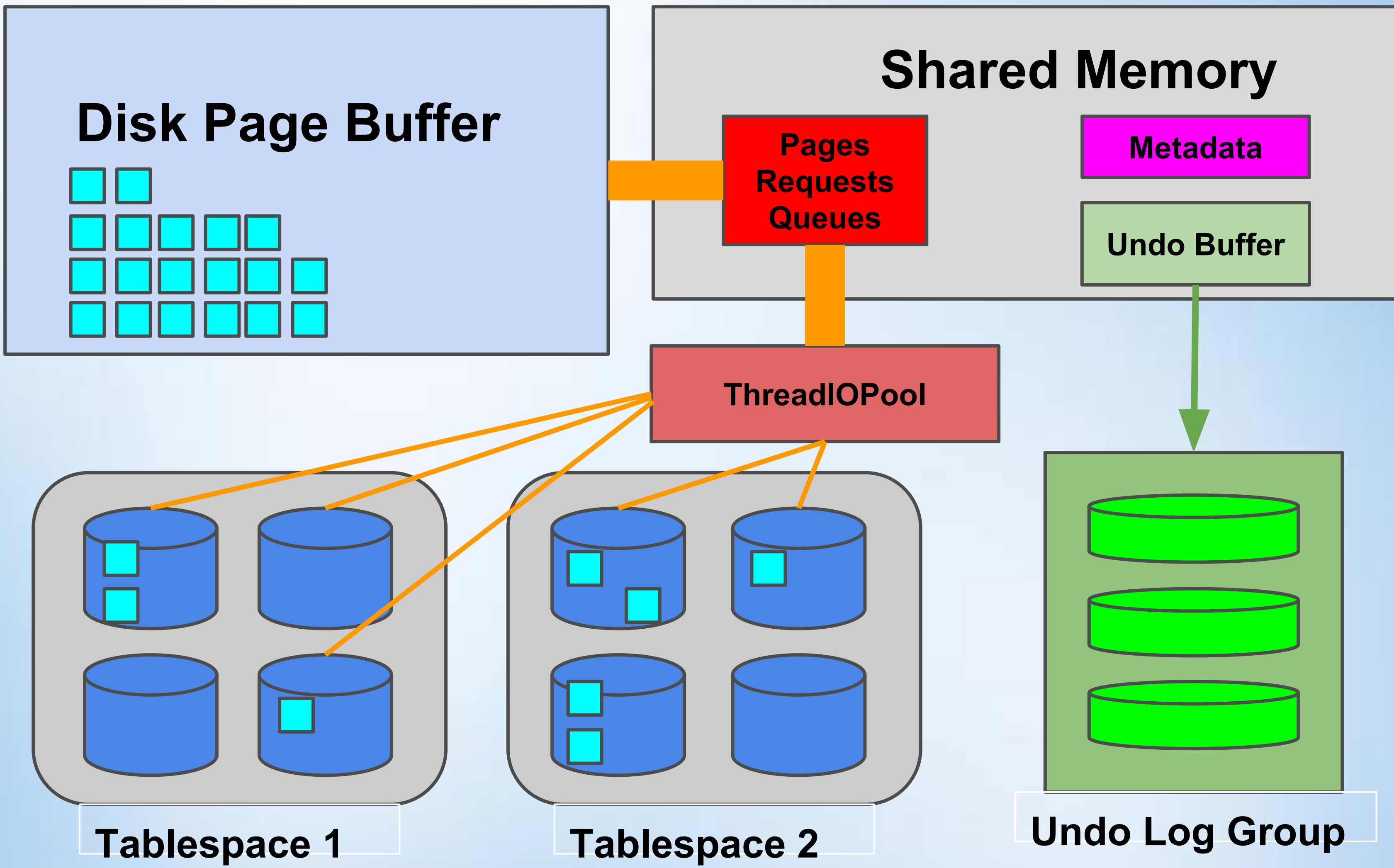
Objects:

- Undo log files: store information to rollback transaction
- Undo log group: a collection of undo log files
- Data files: store MySQL Cluster Disk Data table data
- Tablespace : a named collection of data files

Notes:

- Each tablespace needs an undo log group assigned
- currently, only one undo log group can exist in the same MySQL Cluster

Disk Data - data flow



Disk Data Configuration Parameters

DiskPageBufferMemory : cache for disk pages

SharedGlobalMemory : shared memory for

- DataDisk UNDO buffer
- DiskData metadata (tablespace / undo / files)
- Buffers for disk operations

DiskIOThreadPool : number of threads for I/O

Create Undo Log Group

Using SQL statement:

```
CREATE LOGFILE GROUP logfile_group
    ADD UNDOFILE 'undo_file'
    [INITIAL_SIZE [=] initial_size]
    [UNDO_BUFFER_SIZE [=] undo_buffer_size]
    ENGINE [=] engine_name
```

Create Undo Log Group

UNDOFILE : filename on disk

INITIAL_SIZE : 128MB by default

UNDO_BUFFER_SIZE : 8MB by default

ENGINE : { NDB | NDBCLUSTER }

Alter Undo Log Group

```
ALTER LOGFILE GROUP logfile_group
    ADD UNDOFILE 'undo_file'
    [INITIAL_SIZE [=] initial_size]
    ENGINE [=] engine_name
```

Create Undo Log Group : example

```
CREATE LOGFILE GROUP lg1
    ADD UNDOFILE 'undo1.log'
    INITIAL_SIZE = 134217728
    UNDO_BUFFER_SIZE = 8388608
    ENGINE = NDBCLUSTER

ALTER LOGFILE GROUP lg1
    ADD UNDOFILE 'undo2.log'
    INITIAL_SIZE = 134217728
    ENGINE = NDBCLUSTER
```

Create Undo Log Group

In [NDBD] definition:

InitialLogFileGroup = [name=name;]
    [undo_buffer_size=size;] file-specification-list

Example:

InitialLogFileGroup = name=lg1;
    undo_buffer_size=8M; undo1.log:128M;undo2.log:128M

Create Tablespace

Using SQL statement:

```
CREATE TABLESPACE tablespace_name
    ADD DATAFILE 'file_name'
    USE LOGFILE GROUP logfile_group
    [INITIAL_SIZE [=] initial_size]
    ENGINE [=] engine_name
```

Create Tablespace

logfile_group : mandatory! The log file group must be specified

DATAFILE : filename on disk

INITIAL_SIZE : 128MB by default

ENGINE : { NDB | NDBCLUSTER }

Alter Tablespace

Using SQL statement:

```
ALTER TABLESPACE tablespace_name
    {ADD|DROP} DATAFILE 'file_name'
    [INITIAL_SIZE [=] size]
    ENGINE [=] engine_name
```

Create Tablespace : example

```
CREATE TABLESPACE ts1
    ADD DATAFILE 'datafile1.data'
    USE LOGFILE GROUP lg1
    INITIAL_SIZE = 67108864
    ENGINE = NDBCLUSTER;

ALTER TABLESPACE ts1
    ADD DATAFILE 'datafile2.data'
    INITIAL_SIZE = 134217728;
    ENGINE = NDBCLUSTER;
```

Create Tablespace

In [NDBD] definition:

InitialTablespace = [name=name;] [extent_size=size;]
    file-specification-list

Example:
InitialTablespace = name=ts1;
    datafile1.data:64M; datafile2.data:128M

Note: no need to specify the Undo Log Group

Start Types
and
Start Phases

Start Types

- (IS) Initial start
- (S) System restart
- (N) Node restart
- (IN) Initial node restart

Initial start (IS)

All the data nodes start with clean filesystem.

During the very first start, or when all data nodes are started with --initial

Note: Disk Data files are not removed from the filesystem when data node is started with --initial

System restart (S)

The whole Cluster is restarted after it has been shutdown .

The Cluster resumes operations from where it was left before shutdown. No data is lost.

Node restart (N)

The Node is restarted while the Cluster is running.

This is necessary during upgrade/downgrade, changing configuration, performing HW/SW maintenance, etc

Initial node restart (IN)

Similar to node restart (the Cluster is running) , but the data node is started with a clean filesystem.

Use **--initial** to clean the filesystem .

Note: --initial doesn't delete DiskData files

Start Phases (1/4)

Phase -1 : setup and initialization

Phase 0 : filesystem initialization

Phase 1 : communication inter-blocks and between nodes is initialized

Phase 2 : status of all nodes is checked

Phase 3 : DBLQH and DBTC setup communication between them

Start Phases (2/4)

Phase 4:

- IS or IN : Redo Log Files are created
- S : reads schemas, reads data from last Local Checkpoint, reads and apply all the Global Checkpoints from Redo Log
- N : find the tail of the Redo Log

Start Phases (3/4)

Phase 5 : for IS or S, a LCP is performed, followed by GCP

Phase 6 : node groups are created

Phase 7 : arbitrator is selected, data nodes are marked as Started, and API/SQL nodes can connect

Phase 8 : for S , all indexes are rebuilt

Phase 9 : internal variables are reset

Start Phases (4/4)

Phase 100 : (obsolete)

Phase 101 :

- data node takes responsibility for delivery its primary data to subscribers
- for IS or S : transaction handlers is enabled
- for N or IN : new node can act as TC

Storage requirements

Storage requirements

In general, storage requirements for MySQL Cluster is the same of storage requirements for other storage engine.

But a lot of extra factors need to be considered when calculating storage requirements for MySQL Cluster tables.

4-byte alignment

Each individual column is 4-byte aligned.

```
CREATE TABLE ndb1 (
  id INT NOT NULL PRIMARY KEY,
  type_id TINYINT, status TINYINT,
  name BINARY(14),
  KEY idx_status ( status ), KEY idx_name ( name )
) ENGINE = ndbcluster;
```

Bytes used per row: 4+4+4+16 = 28 (storage)

NULL values

If table definition allows NULL for some columns, MySQL Cluster reserves 4 bytes for 32 nullable columns (or 8 bytes for 64 columns)

```
CREATE TABLE ndb1 (
  id INT NOT NULL PRIMARY KEY,
  type_id TINYINT, status TINYINT,
  name BINARY(14),
  KEY idx_status ( status ), KEY idx_name ( name )
) ENGINE = ndbcluster;
```

Bytes used per row: 4 (NULL) + 28 (storage)

DataMemory overhead

Each record has a 16 bytes overhead

Each ordered index has a 10 bytes overhead per record

Each page is 32KB , and each record must be stored in just one page

Each page has a 128 byte page overhead

IndexMemory overhead

Each primary/unique key requires 25 bytes for the hash index plus the size of the field

8 more bytes are required if the size of the field is larger than 32 bytes

Storage requirement for test table

```
CREATE TABLE ndb1 (
  id INT NOT NULL PRIMARY KEY,
  type_id TINYINT, status TINYINT,
  name BINARY(14),
  KEY idx_status ( status ), KEY idx_name ( name )
) ENGINE = ndbcluster;
```

Memory used per row = ~107 bytes

- 4 bytes for NULL values
- 28 bytes for storage
- 16 bytes for record overhead
- 30 bytes for ordered indexes (3)
- 29 bytes for primary key index (25 bytes + 4 bytes)

plus all the wasted memory in page overhead/alignment

Hidden Primary Key

In MySQL Cluster every table **requires** a primary key.

A hidden primary key is create if PK is not explicitly defined

The hidden PK uses 31-35 bytes per record

Backups

Why backup?

- Isn't MySQL Cluster fully redundant?
- Human mistakes
- Application bugs
- Point in time recovery
- Deploy of development/testing envs
- Deploy a DR site
- ... others

Backup methodologies

mysqldump

NDB Native

Backup with mysqldump

storage engine agnostic

dumps other objects like triggers, view, SP

connects to any API node

Backup with mysqldump (cont.)

Usage:
mysqldump [options] db_name [tbl_name ...]
mysqldump [options] --databases db_name ...
mysqldump [options] --all-databases

```
shell> mysqldump world > world.sql
```

Question: What about `--single-transaction` ?

Backup with mysqldump (cont.)

Bad news:

No write traffic to MySQL Cluster

- SINGLE USER MODE
- reduced availability

Backup with mysqldump (cont.)

Check the content of the dump for:

- CREATE TABLE
- INSERT INTO
- CREATE LOGFILE
- CREATE TABLESPACE

Restore from mysqldump

Standard recovery procedure:

```
shell> mysql -e "DROP DATABASE world;"
shell> mysqladmin create world
shell> cat world.sql | mysql world
```

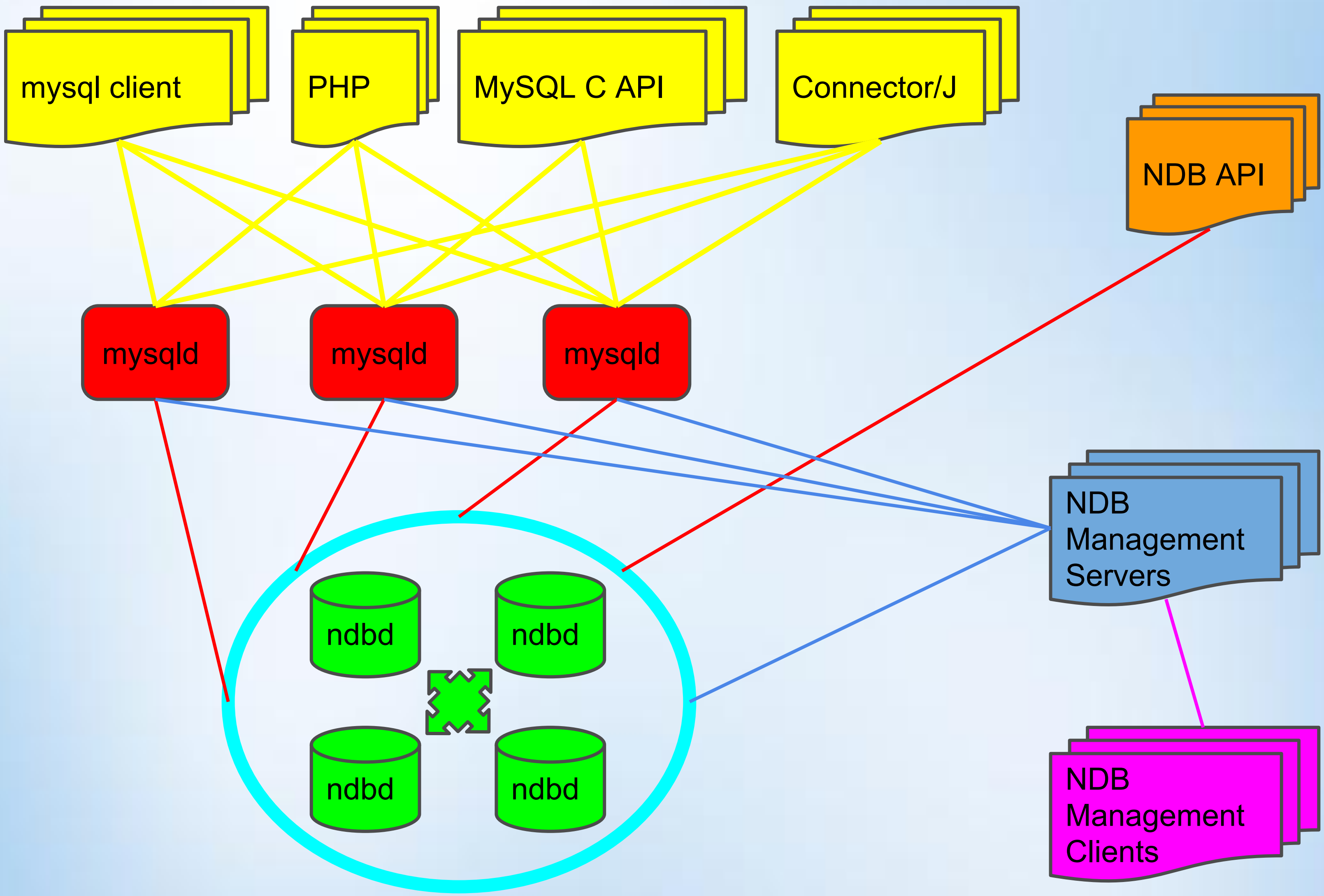
Restore from mysqldump (cont.)

Point in time recovery?

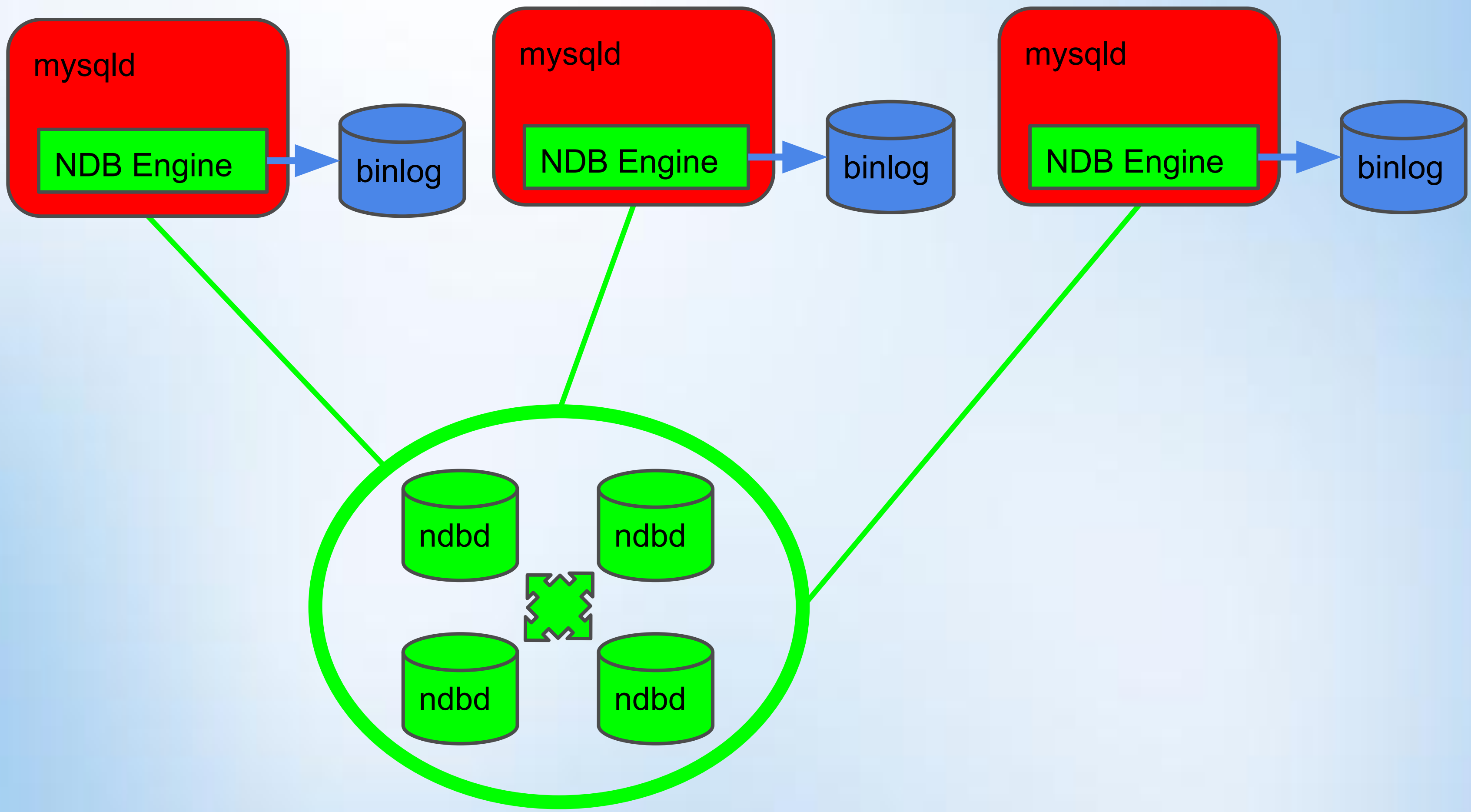
- shell> mysqldump --master-data ...
- apply binlog : from which API node?

NDB and binlog

MySQL Cluster Overview (reminder)



MySQL Cluster and binlog



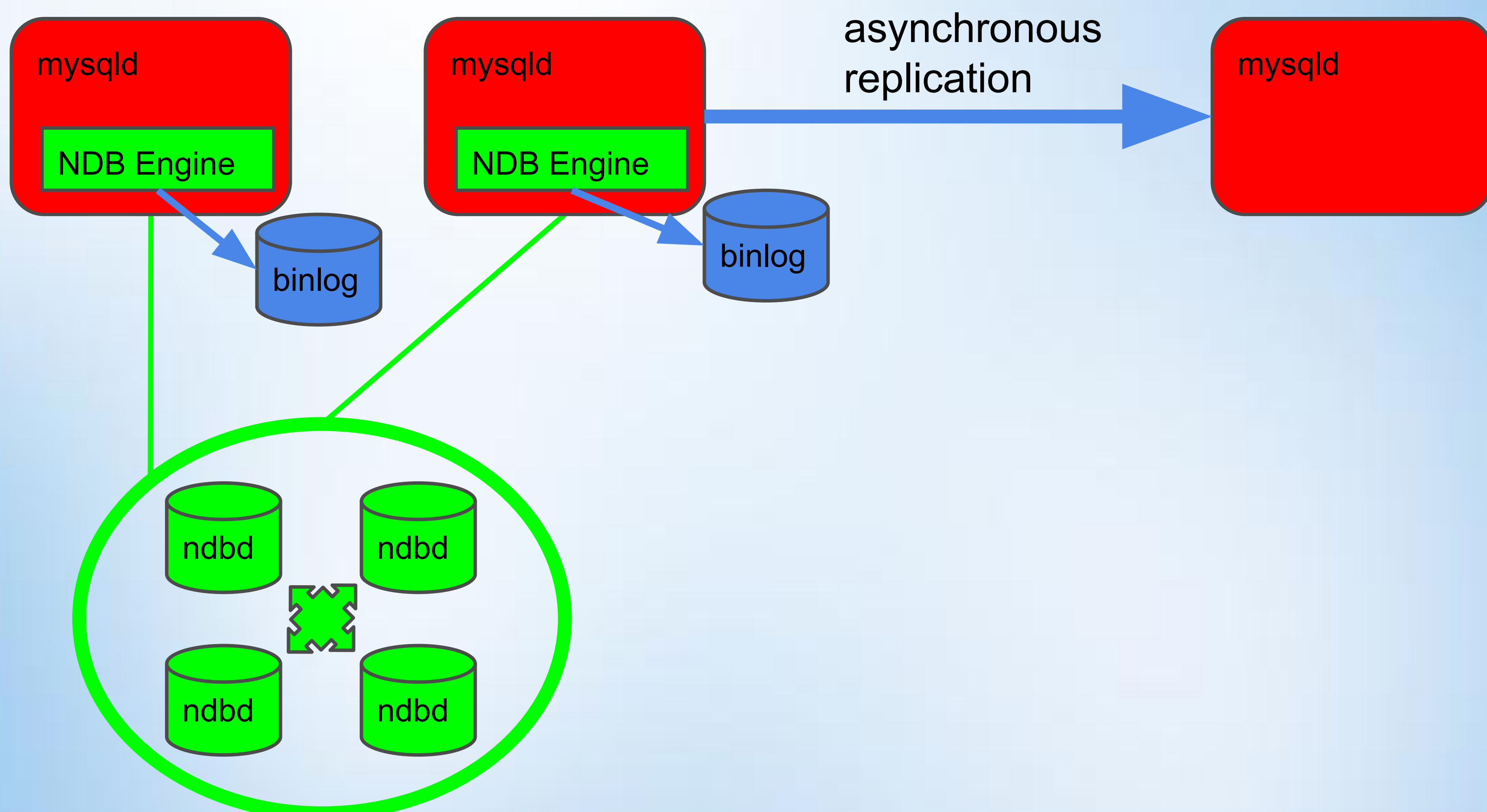
NDB binlog injector thread

When connecting to the data nodes, NDBCluster storage engine subscribes to all the tables in the Cluster.

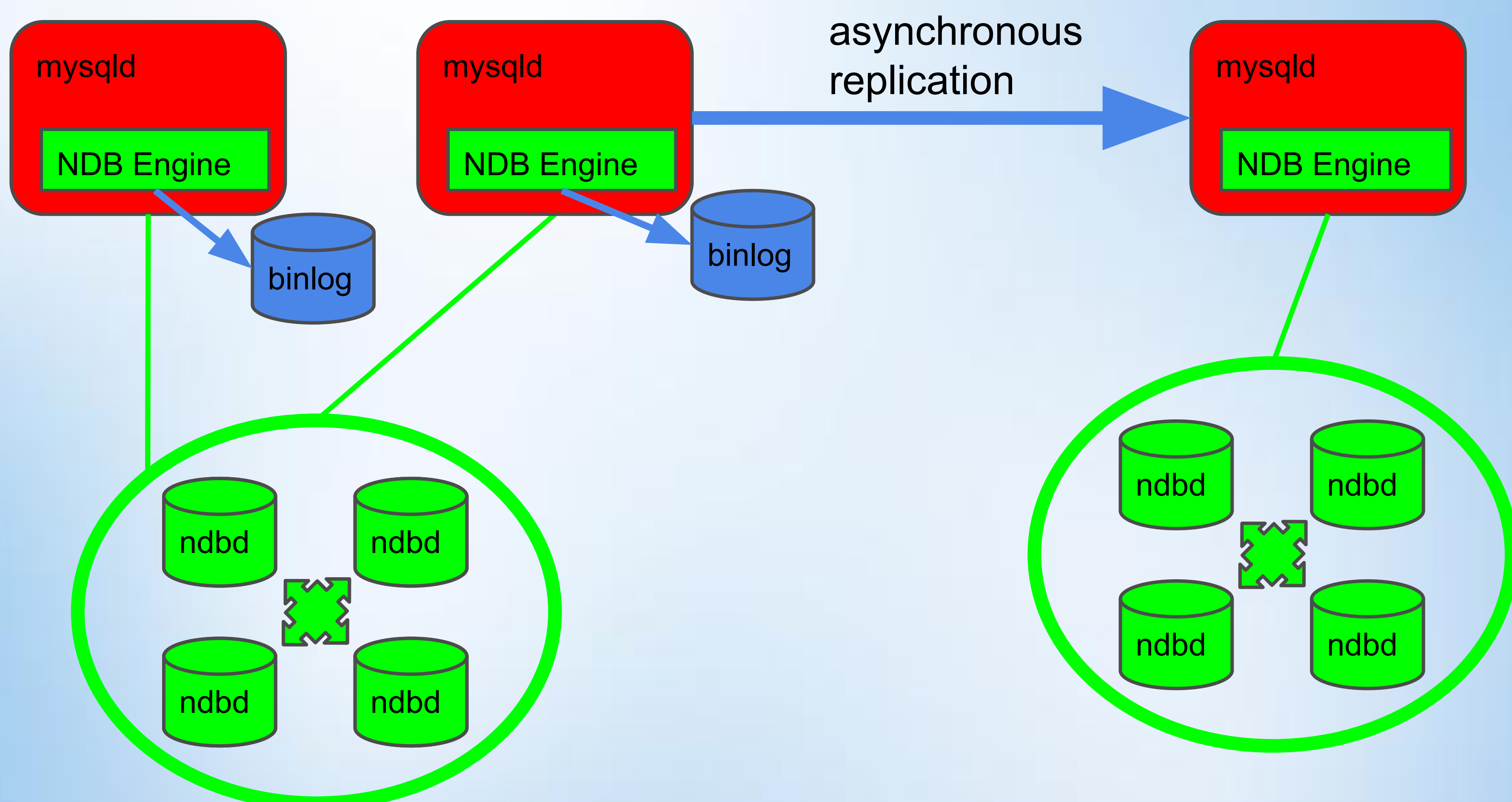
When any data changes in the Cluster, the NDB binlog injection thread get notified of the change, and write to the local binlog.

binlog_format=ROW

Asynchronous Replication to standard MySQL (ex: InnoDB)



Asynchronous Replication to MySQL Cluster



Native Online NDB Backups

Native Online NDB Backup

Hot backup

Consistent backup

Initialized by a cluster management node

NDB Backup Files

Three main components:

- **Metadata** : BACKUP-*backup_id.node_id*.ctl
- **Table records** : BACKUP-*backup_id*-0.*node_id*.data

  different nodes save different fragments during the backup
- **Transactional log** : BACKUP-*backup_id*-0.*node_id*.log

  different nodes save different log because they store different fragments

Saved on all data nodes: each data node performs the backup of its own fragment

START BACKUP

```
shell> ndb_mgm
ndb_mgm> START BACKUP
```

Waiting for completed, this may take several minutes
Node 1: Backup 4 started from node 12
Node 1: Backup 4 started from node 12 completed
StartGCP: 218537 StopGCP: 218575
#Records: 2717875 #LogRecords: 167
Data: 1698871988 bytes Log: 99056 bytes

START BACKUP (cont.)

Waiting for completed, this may take several minutes
Node *Did*: Backup *Bid* started from node *Mid*
Node *Did*: Backup *Bid* started from node *Mid* completed
 StartGCP: 218537 StopGCP: 218575
 #Records: 2717875 #LogRecords: 167
 Data: 1698871988 bytes Log: 99056 bytes

Did : data node starting the backup
Bid : unique identifier for the current backup
Mid : management node coordinating the backup

Number of Global Checkpoints executed during backup
Number and size of records in backup
Number and size of records in log

START BACKUP (cont.)

START BACKUP options:

- NOWAIT
- WAIT STARTED
- WAIT COMPLETED (default)

Backup Parameters

BackupDataBufferSize:
stores data before is written to disk

BackupLogBufferSize:
buffers log records before are written to disk

BackupMemory:
BackupDataBufferSize + BackupLogBufferSize

Backup Parameters (cont.)

BackupWriteSize:

default size of messages written to disk from backup buffers (data and log)

BackupMaxWriteSize:

maximum size of messages written to disk

CompressedBackup:

equivalent to gzip --fast

Restore from Online backup

ndb_restore: program that performs the restore

Procedure:

- import the metadata from just one node
- import the data from each node

ndb_restore

Important options:

-c *string* : specify the connect string

-m : restore metadata

-r : restore data

-b # : backup ID

-n # : node ID

-p # : restore in parallel

--no-binlog

ndb_restore (cont.)

--include-databases = db_lists

--exclude-databases = db_lists

--include-tables = table_lists

--exclude-tables = table_lists

--rewrite-database = olddb,newdb

... and more!