



InnoDB: Architecture, Status, and New Features

Heikki Tuuri
CEO Innobase Oy
Vice President, Development
Oracle Corporation

Ken Jacobs
Vice President, Product Strategy
Server Technologies Division,
Manager, InnoDB group
Oracle Corporation

INNOBASE





Today's Topics

- InnoDB Architecture Overall
- Locking and Concurrency Control
- Forthcoming New Features - Capabilities and Performance
 - Table compression
 - Fast Index creation
- Q & A

InnoDB - Transactions and Reliability for MySQL



- Full transaction support, high concurrency
- High performance memory and i/o architecture
- Flexible storage options
- Efficient indexing and declarative referential integrity to protect foreign-key relationships
- Online backup with InnoDB Hot Backup

INNOBASE



InnoDB Architecture

- Modeled on Gray & Reuter's text *Transaction Processing: Concepts & Techniques*
- Also emulates the Oracle architecture
 - Multi-version concurrency control for higher throughput
 - Undo info in the database, not the logs
 - Tablespaces for data, index storage

INNOBASE

InnoDB Architecture

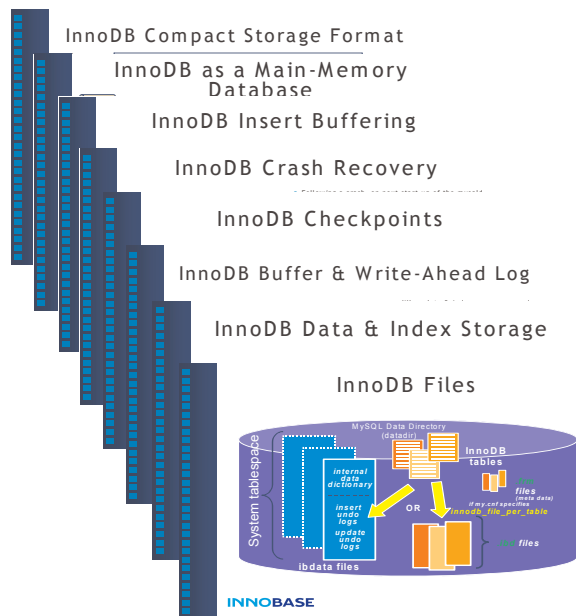
- Today: In-depth on locking and concurrency control
- Previously: overview of many aspects of InnoDB
- See the presentation here:

<http://www.innodb.com>

INNODB > INFO >

InnoDB Overview

2006 MySQL User Conference



INNODB



InnoDB Transaction Model and Locking

- Combines the best properties of multi-versioning with traditional two-phase locking
- InnoDB locks at the row level
- InnoDB runs queries as non-locking consistent reads by default, in the style of Oracle

Typically, users can lock every row in the database, or any random subset of the rows, without InnoDB running out of memory

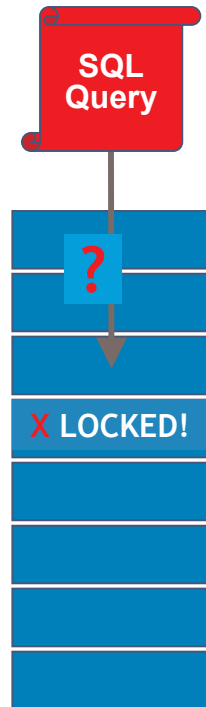
INNOBASE



InnoDB Transactions

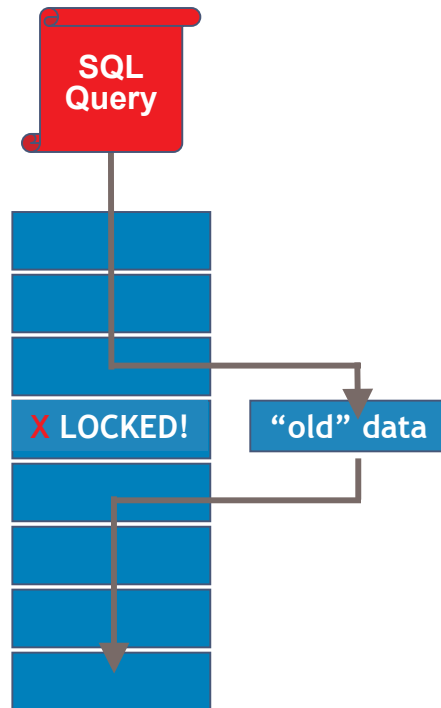
- **A**tomic - all changes are either committed as a group, or all are rolled back as a group
- **C**onsistent - transactions operate on a consistent view of the data, leaving the data in a consistent state (by transaction's end)
- **I**solated - each transaction “thinks” it is running by itself - effects of other transactions are invisible until it commits
- **D**urable - once committed, all changes persist, even if there are system failures

InnoDB Transactions & Locking



- Full transaction support
 - atomicity, consistency, isolation, durability
 - SQL-standard isolation levels
- Unlimited row-level locking
- Multi-version read-consistency
- Automatic deadlock detection

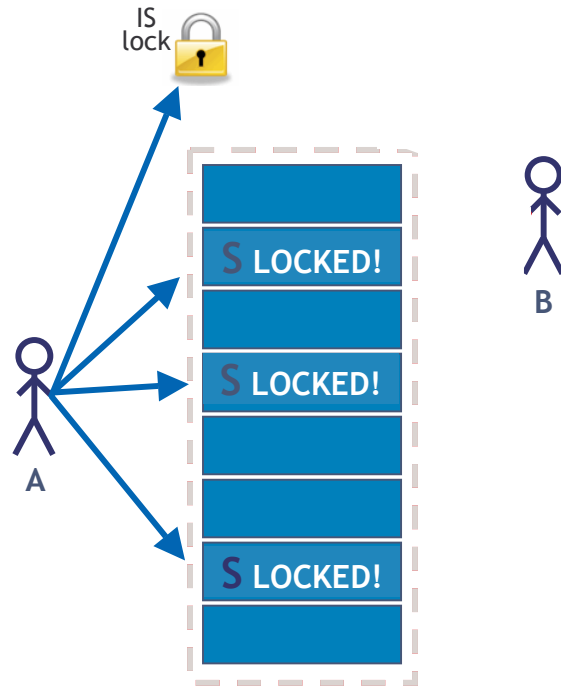
InnoDB Consistent Reads



- Queries see a snapshot of the data consistent with the other data they read
- By default, InnoDB uses “consistent read” for queries like this

```
SELECT a FROM t WHERE b = 2;
```
- Normal undo is used to generate consistent data for the query to see
 - No overhead: undo info is required to rollback uncommitted transactions
- No need to set locks, as history cannot change

Shared and Exclusive Locks



Q: If user A has shared row locks in table T, how does InnoDB know to not let user B set an exclusive X lock on table T?

A: 'Intention locks'. Before setting a shared lock on a row in t, user A sets an 'intention lock' IS on table t.

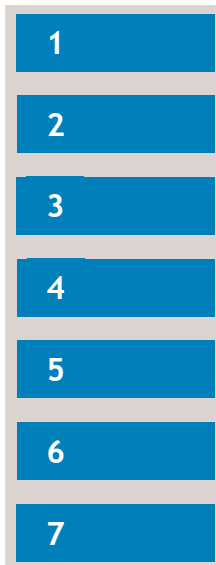
IS is compatible with S, but not with X. Thus, if a user has row share locks, no other user can lock the table in X mode

Similarly, before setting an exclusive X lock on a row, a user must set an IX lock on the table.

- IX is not compatible with S lock on T
- if a user has IX lock on table T, no other user can take S lock on T
- if a user has an S lock on table T, no other user can take X locks on rows in T

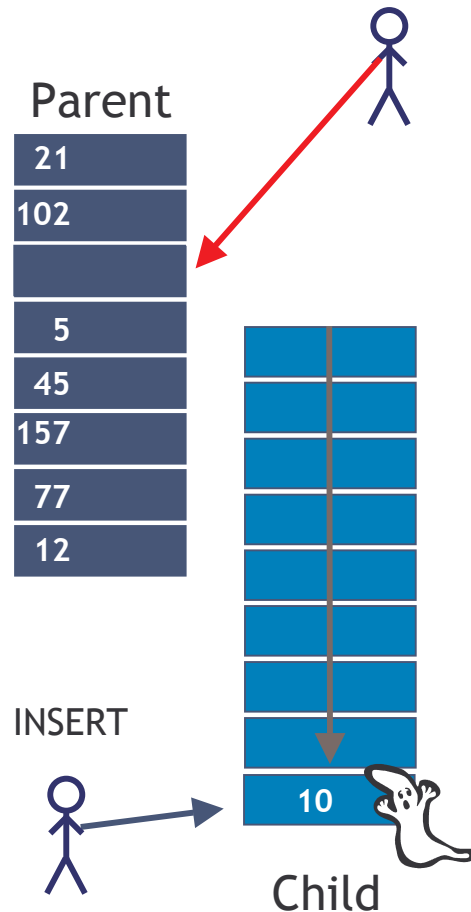
Auto-Increment Locking

- SQL-based logging & replication require auto-increment values to be deterministic
- InnoDB uses a table-level 'auto-increment lock'
 - Table-level lock occurs at time of INSERT
 - Lock is released at statement end, not transaction end
- This table-level auto-increment lock can become a bottleneck in high-concurrency (> 10 threads) INSERT workloads
- We are working on a change to reduce the lock duration for simple single-row and multi-row INSERT statements without a subquery



INNOBASE

Phantoms vs. Consistency



PHANTOM: A row that appears in a second query that was not in the first

Example: foreign key check in application code

- Check that there are no children with parent id=10:

```
SELECT * FROM child  
WHERE parent_id = 10 FOR UPDATE;
```

- If the SELECT returns 0 rows, then

```
DELETE FROM parent WHERE id = 10;
```

- But before first user COMMITs, another user inserts a child with parent_id = 10 ...

➔ INCONSISTENCY!

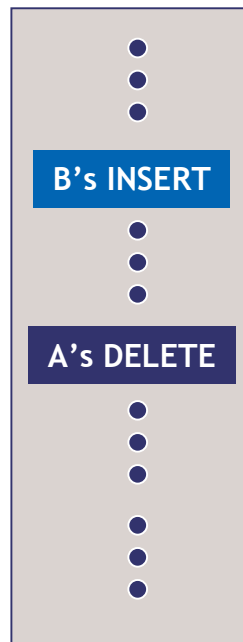
Statement-Based Replication Relies on No Phantoms



```
User A
BEGIN;
DELETE FROM t
  WHERE a = 10;
COMMIT;
```



```
User B
BEGIN;
INSERT INTO t
VALUES (10);
COMMIT;
```



MySQL
binlog

1. User A deletes a row
2. Before A commits, user B inserts the *same* row
3. User B commits before A
4. User A commits
5. The MySQL binlog contain B's transaction before A's

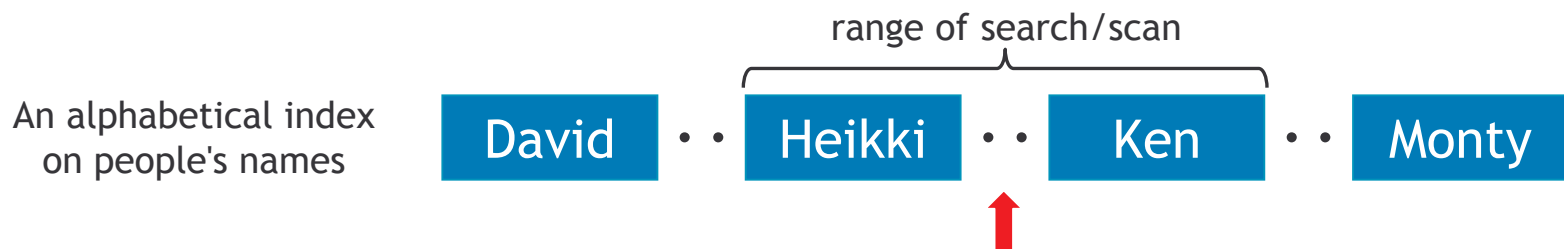
We do not know if A deleted the row that B inserted!



slave may get out-of-sync with the master!

InnoDB Avoids Phantoms Through 'Gap Locking'

- Every SELECT, UPDATE, DELETE in InnoDB uses an index to find the rows to return or operate on
 - ➔ the index is searched, or scanned
- To avoid phantoms, we lock not only the index records we scan, but also the 'gaps' between them
 - ➔ No other user can insert new records in the gaps



If the query scans the rows between Heikki and Ken, we also lock the 'gap' between those records, so that other users cannot insert 'Jeffrey' in the gap

Types of Gap Locking in InnoDB

**Next-key
lock**

locks the key & the gap before the key

**Gap
lock**

locks just the gap before the key

**Record-only
lock**

locks just the key and not the gap

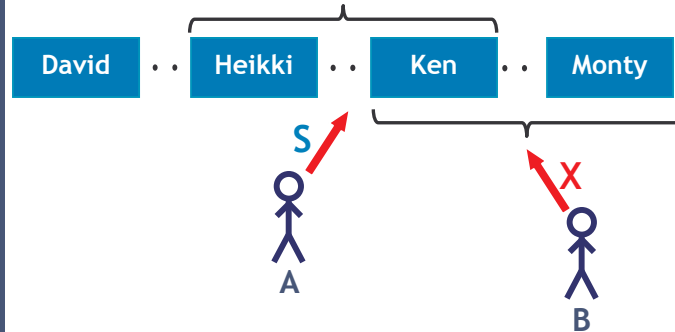
**Insert-
intention
gap lock**

held when waiting to insert into a gap

InnoDB minimizes gap locking by using
record-only locks in UNIQUE searches

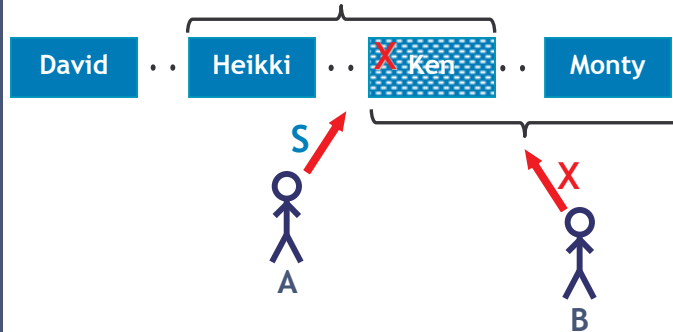
```
UPDATE t SET a = a + 1 WHERE primary_key_value = 100;
```

Gap Locking Subtleties



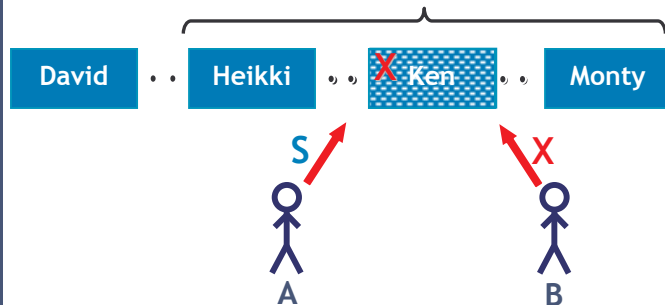
- InnoDB allows two (or more) users to have **conflicting** lock modes on the same gap ...
- Suppose user A locks the gap between HEIKKI and KEN, and B locks the gap between KEN and MONTY

Gap Locking Subtleties



- InnoDB allows two (or more) users to have **conflicting** lock modes on the same gap ...
- Suppose user A locks the gap between HEIKKI and KEN, and B locks the gap between KEN and MONTY
- Now KEN is deleted by user C (who commits)

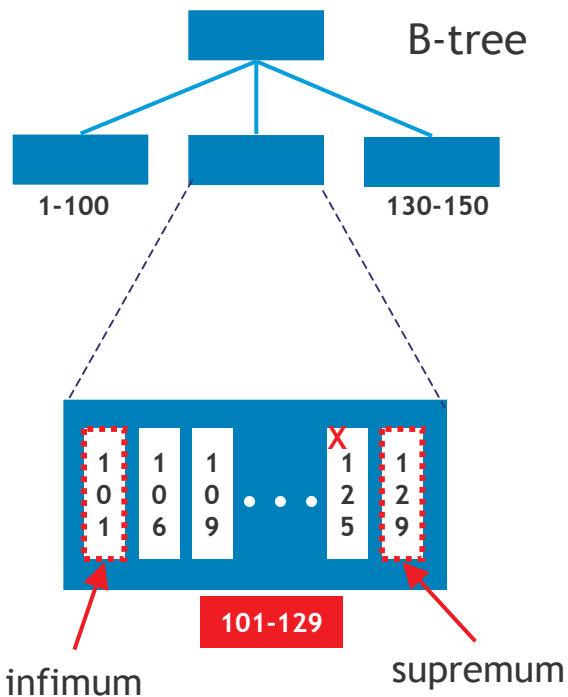
Gap Locking Subtleties



- InnoDB allows two (or more) users to have **conflicting** lock modes on the same gap ...
- Suppose user A locks the gap between HEIKKI and KEN, and B locks the gap between KEN and MONTY
- Now KEN is deleted by user C (who commits)
- User A must respect the lock on the “pseudo-record” that was deleted, but has not yet been purged
- Then the InnoDB purge (garbage collection) thread removes KEN, causing the gaps to merge
- The new, 'bigger' gap inherits the locks from the separate gaps, as well as the lock that was on the deleted record

- InnoDB gap locks are “purely inhibitive” - they block other users from inserting, but do not give the lock owner the privilege to insert into the gap!
 - Even with an exclusive (X) next-key lock on a record R, a user may not be able to insert into the gap before R, since ...
 - Another user might have a gap lock or a “waiting for next-key” lock on R
- The user wishing to insert must wait for conflicting locks to be released

Index Record Locks in InnoDB



infimum and supremum values may not correspond to existing rows, but are the “bounds” of the b-tree node

Can carry locks

- Clustered index (PRIMARY KEY index) and secondary index records
- An ordinary, existing, index record
- A delete-marked index record
- The 'supremum' (least upper bound) pseudo-record on each index page

Cannot carry locks

- The 'infimum' (greatest lower bound) pseudo-record on each index page

This info is displayed in
`SHOW INNODB STATUS \G`

Transaction Isolation Levels

```
SET {SESSION | GLOBAL}  
TRANSACTION ISOLATION LEVEL <level>;
```

SERIALIZABLE

- All plain SELECTs execute as if they used LOCK IN SHARE MODE
- No 'consistent' reads; all SELECTs return the very latest state of the database
- Downside: lots of locking, lots of deadlocks.

REPEATABLE READ

- All SELECTs after the 1st consistent read SELECT in a transaction use the same "snapshot"
- UPDATE, DELETE use next-key locking
- This is the default level

INNOBASE

Transaction Isolation Levels

```
SET {SESSION | GLOBAL}  
TRANSACTION ISOLATION LEVEL <level>;
```

**READ
COMMITTED**

- Each SELECT uses its own “snapshot”
- Data is “up to date”, but multiple SELECTs may be inconsistent with one another
- In V5.1, most gap-locking is removed w/ this level, but you **MUST** use row-based logging/replication
- Fewer gap locks mean fewer deadlocks
- UNIQUE KEY checks in secondary indexes and some FOREIGN KEY checks still need to set gap locks
 - Gaps must be locked to prevent inserting child rows after parent row is deleted
- Many users will move to this isolation level \geq V5.1
- Use `innodb_locks_unsafe_for_binlog` to remove gap locking in MySQL-5.0 and earlier

INNOBASE



Transaction Isolation Levels

```
SET {SESSION | GLOBAL}  
TRANSACTION ISOLATION LEVEL <level>;
```

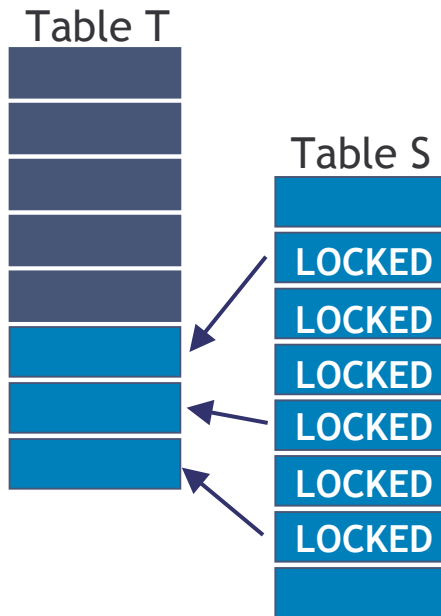
**READ
UNCOMMITTED**

- SELECTS will see data changes made by other users, but not committed
- **NOT RECOMMENDED**: no guarantee of data consistency

INSERTs w/ a SELECT Subquery



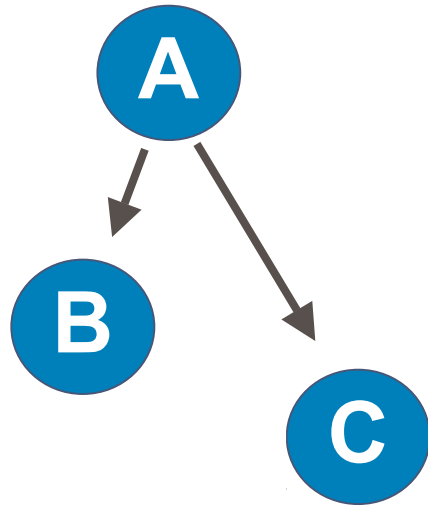
```
INSERT INTO t  
SELECT ...  
FROM s ...
```



- Using REPEATABLE READ (the default), InnoDB locks all records it scans in table S with share locks
- This is to prevent phantoms, which would break MySQL statement-based replication
- **Problem:** this type of query is often used in report generation
- **Fix:** use MySQL V5.1 and READ COMMITTED
 - Then table S is read using a consistent, non-locking read
 - Note: if the result in t is used in replication, then you **MUST** use row-based replication.

INNOBASE

Deadlock Detection & Rollback

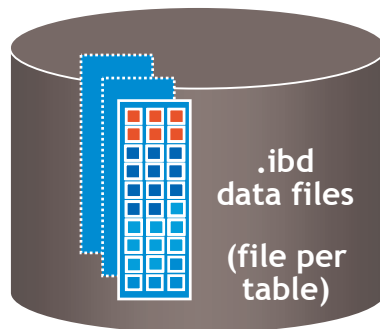
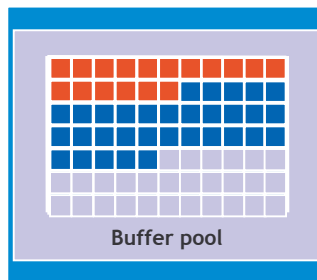


*waits-for
graph*

INNOBASE

- InnoDB automatically detects deadlocks if it detects a cycle in “waits-for” graph of transactions
- Given a deadlock, InnoDB chooses the transaction that modified the fewest rows as the victim, and rolls it back
- Note: InnoDB cannot detect deadlocks that span MySQL storage engines
 - Set `innodb_lock_wait_timeout` in `my.cnf`, to break deadlocks via timeout (default 50 sec)

Transparent Table Compression



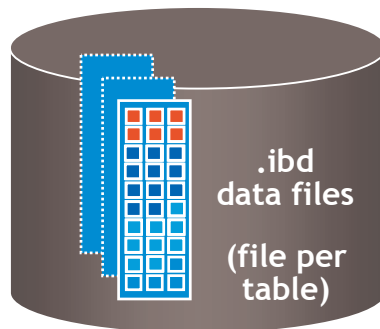
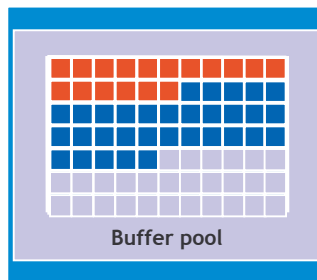
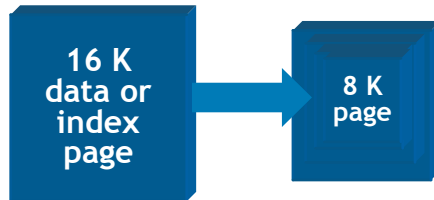
INNOBASE

- The normal InnoDB page size is 16kB
- Specify table's compressed page size (4 or 8 kB)

```
CREATE TABLE t ...  
    ROW_FORMAT=COMPRESSED BLOCK SIZE = 4K
```

- InnoDB uses LZIP to transparently compress data & index pages to the specified size
- Works for all data, not just read-only
- Saves buffer pool memory & disk space too
 - Buffer pool allocation for compressed pages is automatic (using LRU), with manual override
 - Under some disk-bound workloads, it may make sense to allocate 90 % of the buffer pool for compressed pages.
- Requires "file per table"
(my.cnf option innodb_file_per_table)

Table Compression Performance



INNOBASE

- Test table definition

```
create table zip1
```

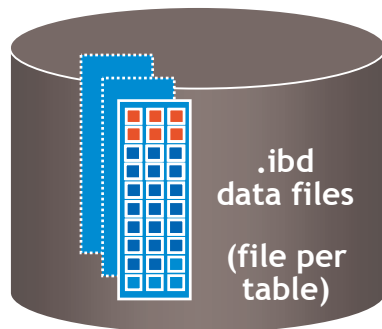
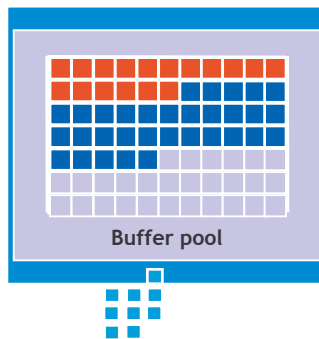
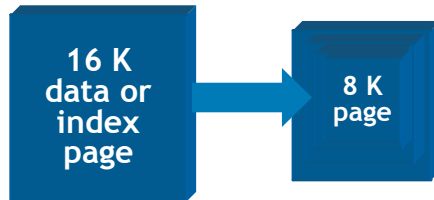
```
(a int not null auto_increment,  
 b int, c int, d varchar(255),  
 primary key (a), index (b), index (c),  
 index (d(20)))
```

```
row_format = compressed
```

```
block_size = 8K engine = innodb")
```

- 8,000,000 rows
- 1 GB memory desktop computer
- Disk-bound workload
- Single-user
- Synthetic data

Table Compression Performance



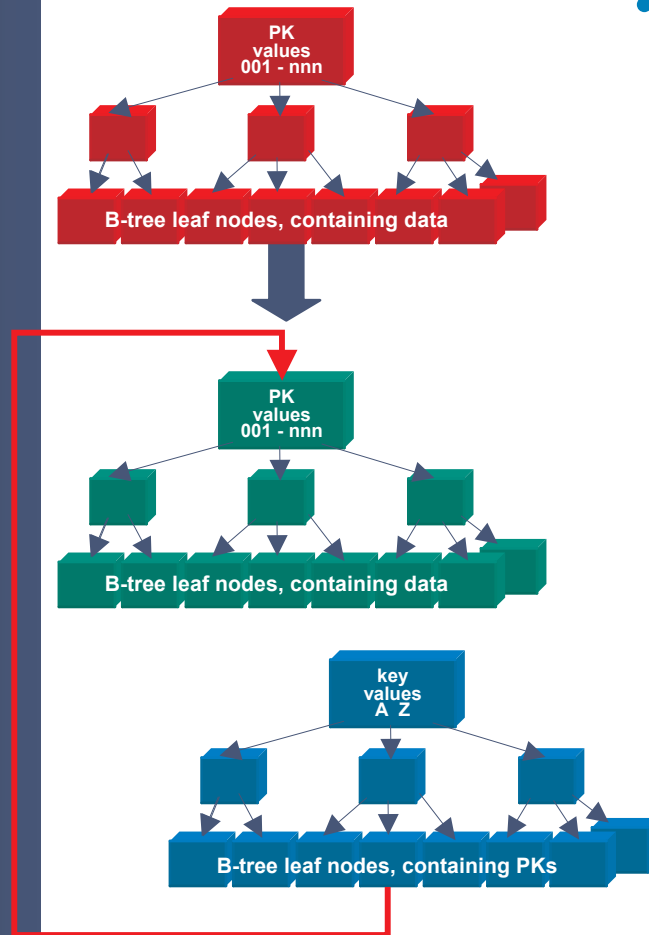
INNOBASE

| | Uncompressed | Compressed |
|----------------|---------------|---------------|
| File size | 2.8 GB | 1.4 GB |
| Inserts | 1300 / second | 1000 / second |
| Table scan | 67 seconds | 90 seconds |
| Random SELECTs | 100 / second | 100 /second |
| CPU usage | 5 to 50 % | 15 to 50% |

With tests using real-world data, compression has reduced file size to 10% of original

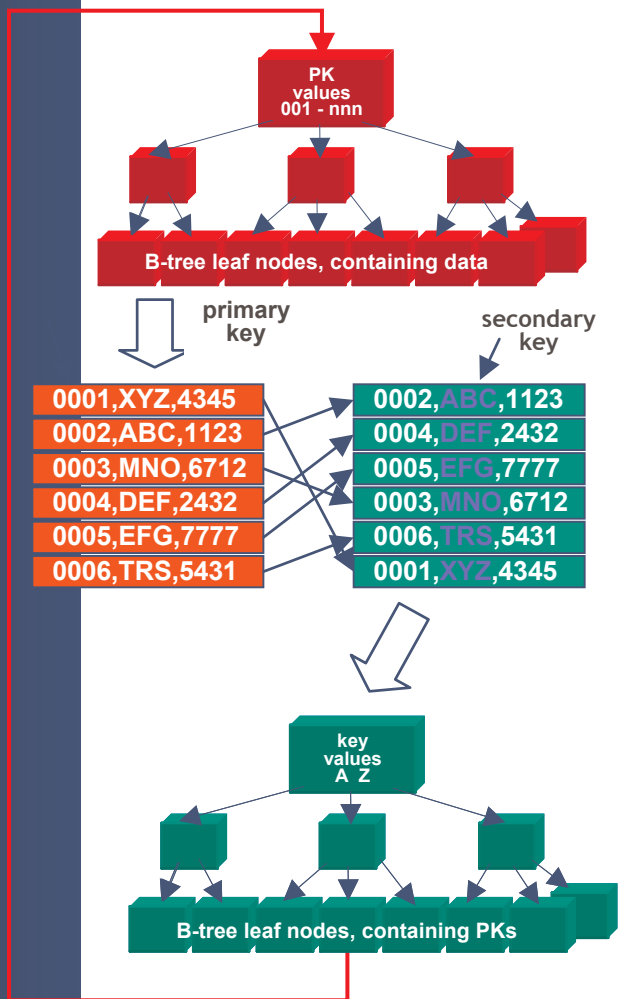
InnoDB Fast Index Create

- MySQL rebuilds an entire table, row-by-row, to create a new secondary index
`CREATE INDEX myindex ON t (col2);`



INNOBASE

InnoDB Fast Index Create



INNOBASE

- MySQL rebuilds an entire table, row-by-row, to create a new secondary index
`CREATE INDEX myindex ON t (col2);`
- MySQL 5.1 allows the storage engine to build just the requested indexes, and not the entire table
- This also applies to `DROP INDEX`
- In MySQL V5.2, InnoDB will sort the table's rows on the secondary key
- Then it will insert the rows into the secondary index
- Creating new indexes for InnoDB tables becomes much faster, because the table is not re-created and because the data are inserted in order



InnoDB Summary

- InnoDB is the leading transactional storage engine for MySQL
- InnoDB's architecture is well-suited to modern, on-line transactional applications
- New InnoDB features for MySQL V5.1 and V5.2 improve performance and convenience
- More new developments underway!

For more info, please visit
www.innodb.com

INNOBASE



INNOBASE

developer of

INNODB[®]

and

INNODB[®] Hot Backup

ORACLE[®]

Innbase is an Oracle company

