PostgreSQL Conference East 2009

The Art of Indexes

Josh Williams

~~PostgreSQL Conference~~ East 2009

JDCon East 2009

The Art of Indexes

Josh Williams

Introduction to Indexes

- All About Performance, avoid this:

```
                          QUERY PLAN
-----------------------------------------------------------
 Seq Scan on table  (cost=0.00..16421.58 rows=1 width=4)
```

- Each index can dramatically improve performance, both read and write
- But maintaning indexes requires a performance hit

The Art of Indexes

- Index Type Overview
- Reading EXPLAIN output
- Multi-column indexes
- Expression-based (functional) indexes
- Partial indexes
- Clustered indexes

The General Syntax

```
CREATE INDEX index_name
   ON table_name ( column1, column2 );
```

The Full Syntax

```
CREATE [UNIQUE] INDEX [CONCURRENTLY]
   index_name ON [schema.]table_name
   [USING index_type]
   ( column1 or (expression1), ... )
   [WITH (FILLFACTOR = value)]
   [TABLESPACE tablespace_name]
   [WHERE predicate]
   ;
```

Reading EXPLAIN

- EXPLAIN shows the planner output
- <query> → `EXPLAIN` <query>
- EXPLAIN ANALYZE runs query, shows run time
  - INSERT, UPDATE, DELETE -will- change the DB
  - ```
    BEGIN;
    EXPLAIN ANALYZE UPDATE ...;
    ROLLBACK;
    ```


    ```

Reading EXPLAIN

```
=# EXPLAIN SELECT * FROM orders WHERE customerid = 1111;
                          QUERY PLAN
-----------------------------------------------------------
 Seq Scan on orders  (cost=0.00..250.00 rows=2 width=36)
   Filter: (customerid = 1111)
(2 rows)

=# CREATE INDEX orders_cid_idx ON orders (customerid);
CREATE INDEX
=# EXPLAIN SELECT * FROM orders WHERE customerid = 1111;
                          QUERY PLAN
-----------------------------------------------------------
 Index Scan using orders_cid_idx on orders
  (cost=0.00..8.27 rows=2 width=36)
   Index Cond: (customerid = 1111)
(2 rows)
```

Reading EXPLAIN

- A larger example...

```
=# EXPLAIN SELECT * FROM customers INNER JOIN orders USING
   (customerid) INNER JOIN orderlines USING (orderid) INNER
   JOIN products USING (prod_id) INNER JOIN categories
   USING (category);
                                                   QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=2811.58..8440.64 rows=60350 width=479)
   Hash Cond: (products.category = categories.category)
   ->  Hash Join  (cost=2810.22..7609.47 rows=60350 width=361)
         Hash Cond: (orderlines.prod_id = products.prod_id)
         ->  Hash Join  (cost=2484.22..5774.72 rows=60350 width=314)
               Hash Cond: (orderlines.orderid = orders.orderid)
               ->  Seq Scan on orderlines  (cost=0.00..958.50 rows=60350 width=18)
               ->  Hash  (cost=1859.22..1859.22 rows=12000 width=300)
                     ->  Merge Join  (cost=0.10..1859.22 rows=12000 width=300)
                           Merge Cond: (customers.customerid = orders.customerid)
                           ->  Index Scan using customers_pkey on customers  (cost=0.00..963.25 rows=20000 width=268)
                           ->  Index Scan using orders_cid_idx on orders  (cost=0.00..696.22 rows=12000 width=36)
         ->  Hash  (cost=201.00..201.00 rows=10000 width=51)
               ->  Seq Scan on products  (cost=0.00..201.00 rows=10000 width=51)
   ->  Hash  (cost=1.16..1.16 rows=16 width=122)
         ->  Seq Scan on categories  (cost=0.00..1.16 rows=16 width=122)
(16 rows)
```

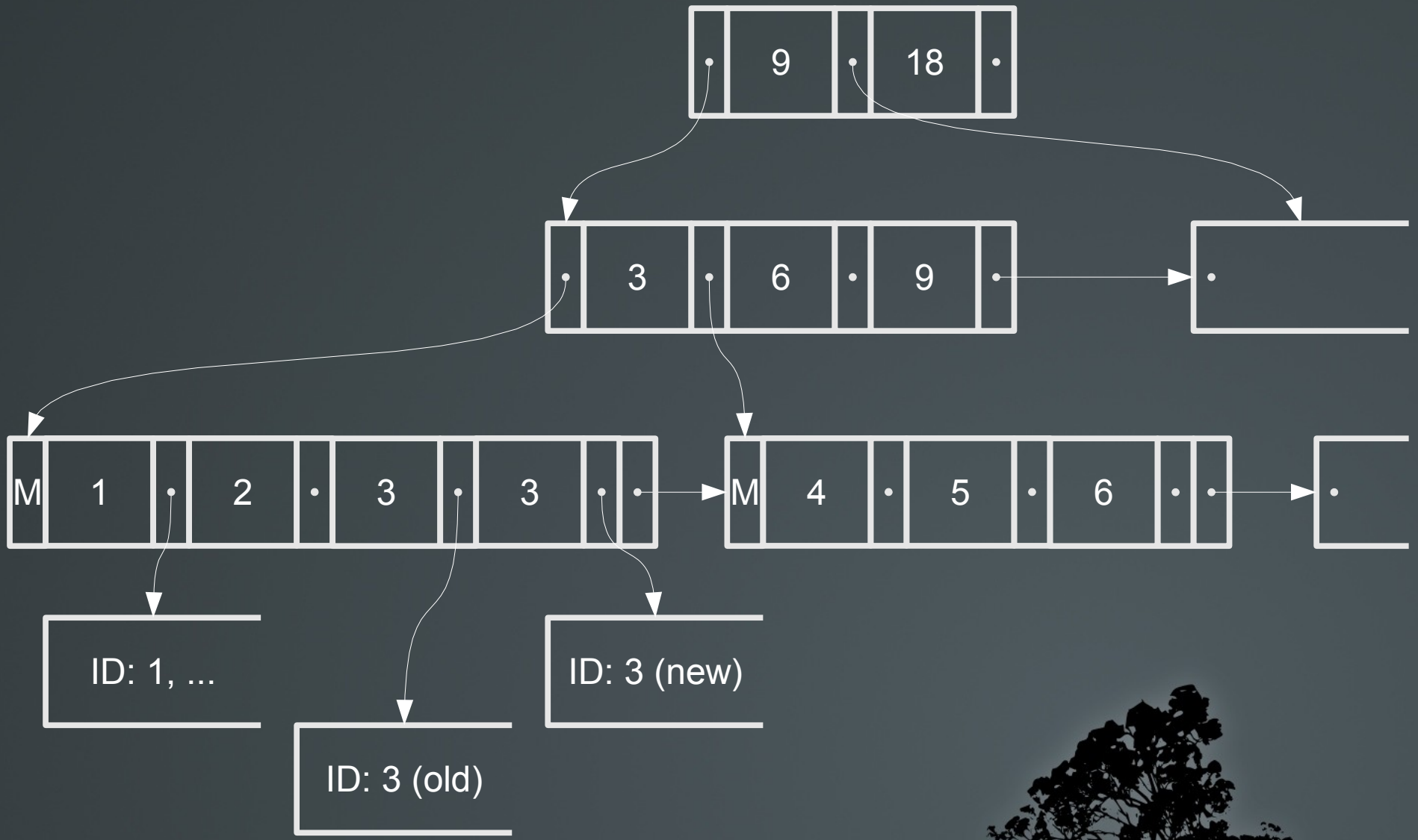
Index Type Overview

- Direct Types:
  - B-tree
  - Hash
- Container Types:
  - GIN
  - GiST

Index Types: B-tree

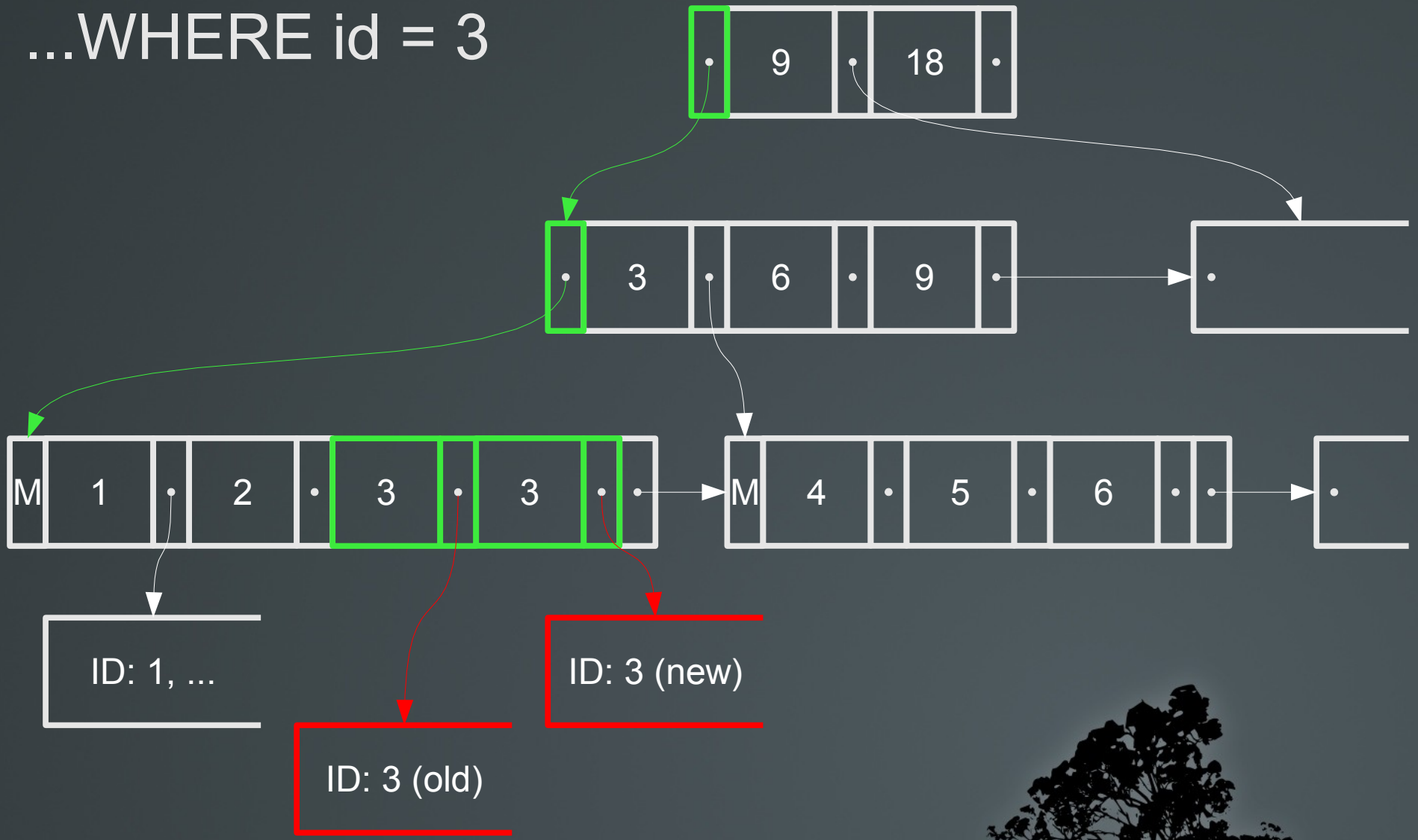
- Default index type
- Strengths
  - Supports range queries on common data types (number & string types)
- Weaknesses
  - Upper limit on field size (~1/3 page size)

Index Types: B-tree



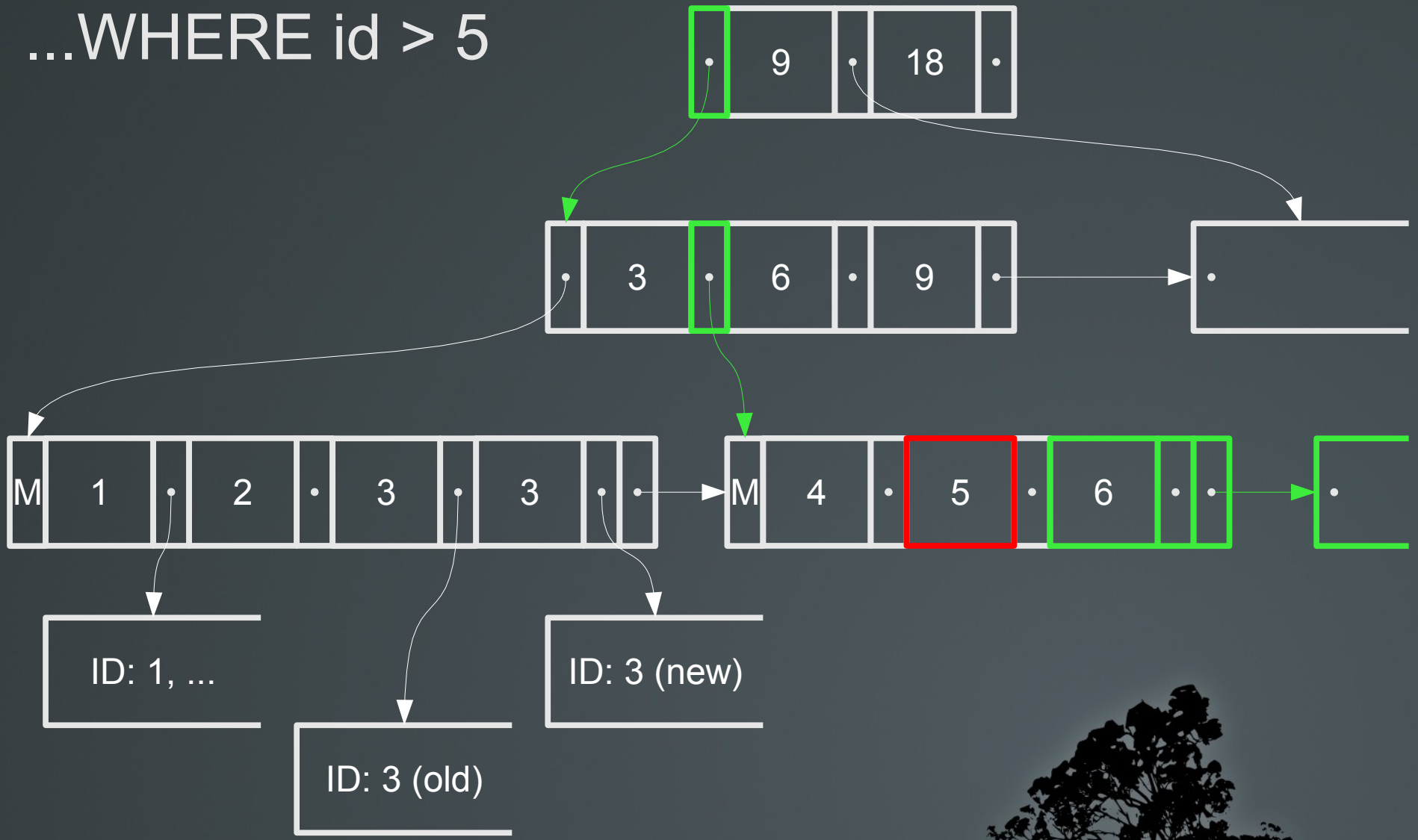
Index Types: B-tree

...WHERE id = 3



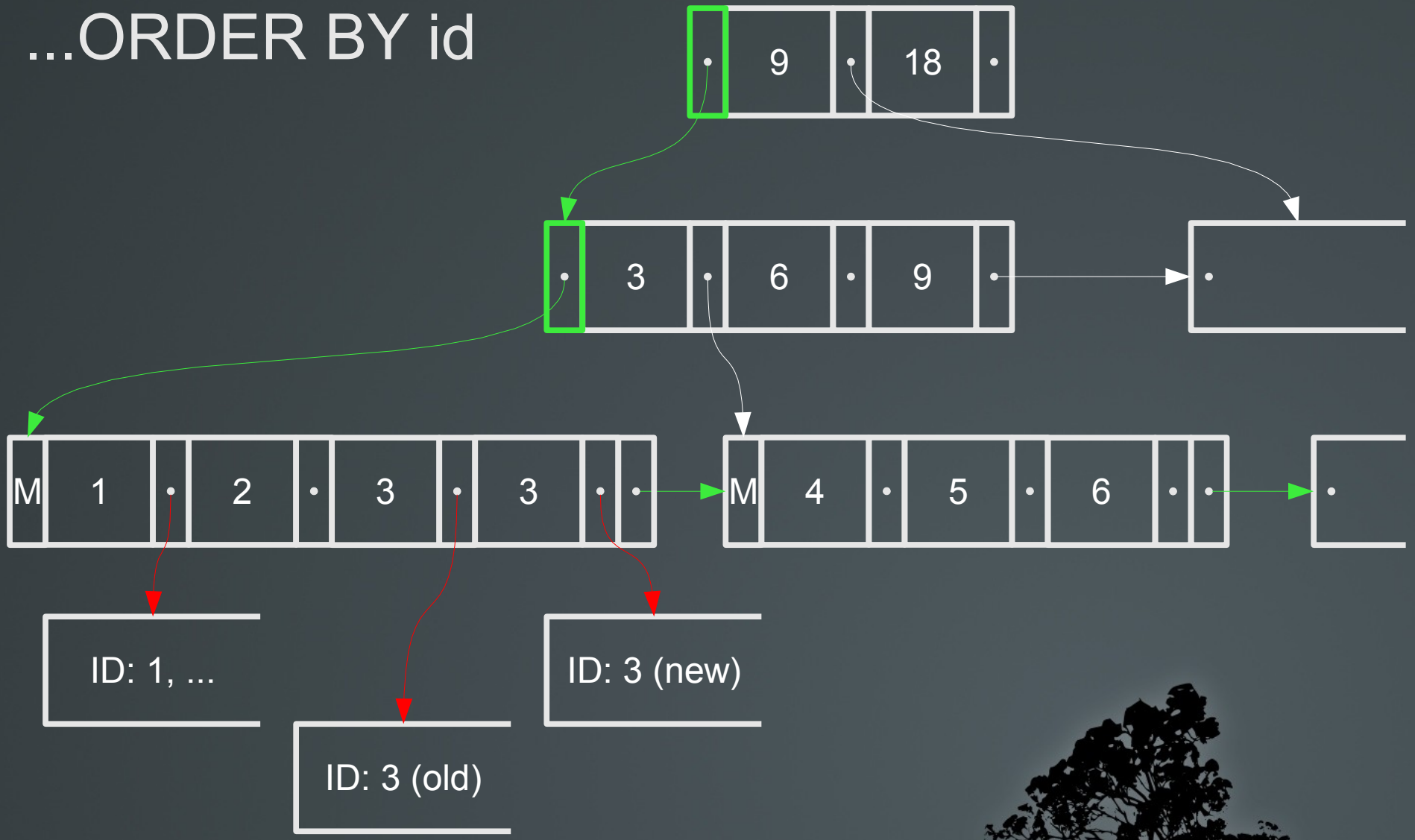
Index Types: B-tree

...WHERE id > 5



Index Types: B-tree

...ORDER BY id



Index Types: Hash

- Strengths
  - Better at handling low-cardinality situations than B-tree
  - Received some improvements in 8.4 (Robert said so.)
- Weaknesses
  - Poor build time, search time about the same as B-tree
  - Only supports equality operator, No mechanism to reduce size, No CLUSTER support, No WAL-logging
- Nah, better off not considering this one

Index Types: GIN

- Generalized Inverted Index
- Useful for array types (including full text columns)
- Similar to hash indexes...
  - Sets of key/'element' and 'posting list'/matching row pairs
  - Multiple rows per key, Multiple keys per row
- Exensible: Users can define functions for their data types to interface with the index
- B-tree (keys), pointing to a B-tree (or list) of tuples

Index Types: GiST

- Generalized Search Tree
- Useful for spatial data types (and full text columns)
- Extensible: Provides a general framework to create tree structures based on user functions
- The output of the user functions define how the tree is created (and searched.)

Index Types: GIN

- Strengths
  - Indexing columns with multiple keys
  - 8.4 will support multiple columns
- Weaknesses
  - Only supports equality (type) operators
  - ... Though 8.4 supports partial matches
  - Can be expensive to construct

Index Types: GiST

- Strengths
  - Very versatile, if you have a custom data type
  - Useful for geometric types (and PostGIS!)
- Weaknesses
  - Tend to be slower to search
  - Not as useful for built-in data types

Full Text Index Strategy

- Both GiST and GIN support full text searches.....
- Which one to use is a matter of trade offs:
  - GiST is always ”lossy” and prone to false index matches
  - But it's fast to maintain!
  - GIN is big and slow, and has an operator quirk
  - But it's faster to query, and supports larger documents!
- The Fine Manual's rule of thumb...
  - Static data (e-books) GIN; Dynamic data (forums) GiST.

Invoking Your New Index

- Run `ANALYZE` on the table – update statistics
- If it still doesn't work, force it to use the index...
  - `SET enable_seqscan = off;`
  - `SET enable_nestloop = off;`
  - `EXPLAIN ANALYZE` (or \timing in psql)
- If it *still* doesn't work, make sure the index definition (including selected expressions or columns, partial index predicate) matches.

Indexing Strategy Decisions

- Simple indexing tricks: Concurrent builds, UNIQUE
- Multiple columns, versus many single-column
- Indexes on Expressions / Functions
- Partial Indexes
- Clustering Indexes

Create Index CONCURRENTLY

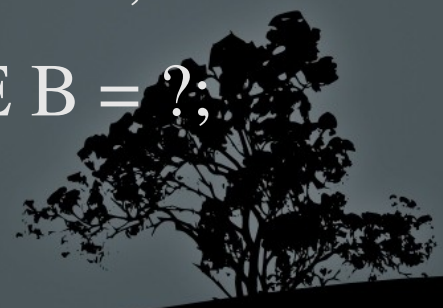
- The index build process: Expensive
- `CREATE INDEX` obtains a table-wide SHARE lock (i.e. all write activity is blocked.)
- `CONCURRENTLY` allows write activity, naturally
  - Performs two scans over the table (separate transactions)
  - Has to wait for previous transactions to finish
  - If it runs into problems (unique, function errors) ... invalid index just hangs around out there
  - `DROP` and try again...

The UNIQUE Index

- (Primary Key) Unique Constraint ==> Unique Index
- Index is checked for existing values
- Remember: NULL != NULL
- (Primary Key) Multiple columns are supported
- Usually considered to be an implementation detail (in other words, use `ADD CONSTRAINT UNIQUE`) ... but there's a couple cool things you can do with it.

Multi-Column Indexes

- Supported by B-tree, GiST, and GIN (8.4)
- Most useful in ”parent/child” column relationships
  - Major, minor device numbers
  - Order number, item number in an order detail table
- An index on columns (A, B)...
  - Really useful for SELECT ... WHERE A = ? AND B = ?;
  - Still useful for SELECT ... WHERE A = ?;
  - Not so useful for SELECT ... WHERE B = ?;



Multiple Single-Column Indexes

- Bitmap Index Scan: use multiple indexes together
- Each index produces a bitmap of tuples (or pages)...
- The bitmaps are AND'ed (or OR'ed) together...

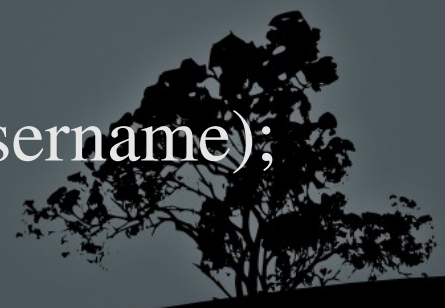
```
Tuple:    1 2 3 4 5 6 7 8
Idx1:     0 1 0 0 1 0 0 1
Idx2:     0 1 0 1 0 0 0 1 &&
Result:   0 1 0 0 0 0 0 1
```

Multiple Single-Column Indexes

- So multi-column indexes are no longer as important.
- An index on column (A) + an index on column (B)...
  - Useful for SELECT ... WHERE A = ?;
  - Useful for SELECT ... WHERE B = ?;
  - Useful for SELECT ... WHERE A = ? AND B = ?;
- But you have to maintain 2 (or more) indexes

Expression-Based Indexes

- Indexes can be built on the result of an expression
- Classic example: Usernames
  - `CREATE INDEX` ... `ON` users (lower(username));
  - `SELECT` ... `FROM` users `WHERE` lower(username) = '?';
- Can be expensive to build (and maintain)
- Any functions used must be marked `IMMUTABLE`
- That cool thing we talked about...
  - `CREATE UNIQUE INDEX` ... lower(username);



The Partial Index

- Restrict what an index will consider
- Similar to `SELECT`'s `WHERE` clause
- `CREATE INDEX` ... `WHERE` balance > 1000;
  - Allow quick access to 'interesting' data, small index
  - The rest of the table accessed through slower methods
- The other cool thing we talked about...
  - `CREATE UNIQUE INDEX` ... (username) `WHERE` active_account = true;

Clustering an Index

- Rewrites a physical table in index order
  - Like `VACUUM FULL` ... but useful!
- A one-time operation, index order is not maintained
  - (But FILLFACTOR can help)
- Needs twice the disk space, and an exclusive lock
- Very useful for range queries...
  - `WHERE created BETWEEN '2009-03-01' and '2009-04-01'`
- Run `ANALYZE` afterward, to update statistics

What Index Options Are Right?

?

The Art of Indexes

- Index Type Overview
- Reading EXPLAIN output
- Multi-column indexes
- Expression-based (functional) indexes
- Partial indexes
- Clustered indexes

The Art of Indexes

Slides...

http://dbahumor.com/indexes