
MONTANA
STATE UNIVERSITY

DDL - Data Definition Language

•••

Indexes, Triggers & Stored Procedures

Last Lecture

- We discussed how to express constraints on data columns in a database

```
CREATE TABLE albums_bak (
    AlbumId  INTEGER NOT NULL PRIMARY KEY,
    Title    NVARCHAR(160) NOT NULL,
    ArtistId INTEGER
);
```

Indexes

- Recall that in a database, a table is a list of rows
- Rows are laid out sequentially on disc
- In SQLite, unless you say otherwise, all rows have an associated *rowid*

```
CREATE TABLE albums(
    AlbumId  INTEGER,
    Title    NVARCHAR(160),
    ArtistId INTEGER
);
```

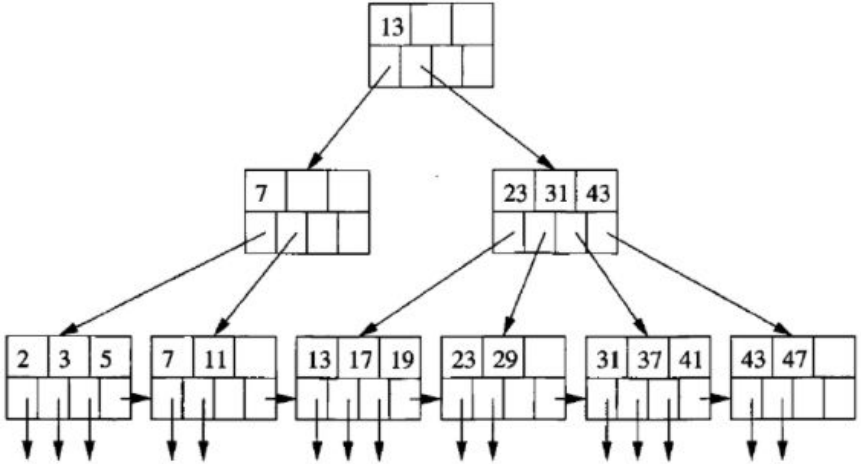
Indexes

- An *index* is a data structure that helps improve the performance of queries
- SQLite uses B-Trees for maintaining indexes
  - B-Tree stands for *Balanced Tree* (not Binary Tree)
  - B-Trees allow for efficient (O(log n)) queries using relational operators

```
CREATE TABLE albums(
    AlbumId  INTEGER,
    Title    NVARCHAR(160),
    ArtistId INTEGER
);
```

Indexes

- We will discuss B-Trees in more detail later in the course
- Note that this B-Tree has pointers between leaf notes, making it a B+ tree
  - Efficient find as well as ordered iteration



Indexes

- Indexes are associated with a specific table
- Indexes are added on specific columns in the table
- You typically add an index for each common access pattern
  - Verified with profiling!

```
CREATE TABLE albums(
    AlbumId  INTEGER,
    Title    NVARCHAR(160),
    ArtistId INTEGER
);
```

Indexes

- Adding an index required the CREATE INDEX statement:

  CREATE [UNIQUE] INDEX <index_name> ON <table_name>(<column_list>)

```
-- email index
CREATE UNIQUE INDEX idx_employees_email
    ON employees (Email);
```

Indexes

- Note that indexes can place UNIQUE constraints on columns
- In a sense, indexes are a specialized constraint put on columns
  - Useful particularly in multi-column indexes discussed in a bit

```
-- email index
CREATE UNIQUE INDEX idx_employees_email
    ON employees (Email);
```

Indexes

- Here we are adding an index on the Email column in the employees table
- EXPLAIN QUERY PLAN before and after demo…

```sql
-- email index
CREATE UNIQUE INDEX idx_employees_email
    ON employees (Email);
```

Indexes

- Note: an index on a text column like this is useful only on:
  - Relational queries
  - Prefix searches
- General text search will still cause a table scan
  - Why?

```
-- email index
CREATE UNIQUE INDEX idx_employees_email
    ON employees (Email);
```

Indexes - Multicolumn

- Consider a query against the playlist_track join table and an associated index
- Is this index helpful?

```sql
SELECT *
FROM playlist_track
WHERE PlaylistId = 1 AND TrackId = 1;

CREATE INDEX idx_playlist_track_1
    ON playlist_track (PlaylistId);
```

Indexes - Multicolumn

- Here is a multi-column index
- This is perfectly tuned for this query
  - O(log(n)) search time
  - No additional scanning
- SQLite is smart enough to add this index automatically because fields are part of the PK
  - Look for indexes named sqlite_autoindex

```sql
SELECT *
FROM playlist_track
WHERE PlaylistId = 1 AND TrackId = 1;

CREATE INDEX idx_playlist_track_2
    ON playlist_track (PlaylistId, TrackId);
```

Indexes - Foreign Keys

- FKs almost always need an index to perform well
- SQLite does *not* automatically add FK indexes, so they must be added *manually*

```
create index IFK_TrackAlbumId
    on tracks (AlbumId);
```

Indexes - Performance

- Much of database tuning is adding and tweaking indexes
- Indexes increase read performance
- However, indexes can hurt insert, update and delete performance
  - DB must now do more work
- Make sure you have empirical data before you start adding!

```
create index IFK_TrackAlbumId
    on tracks (AlbumId);
```

Indexes - Expressions

- SQLite supports *expression indexes*
- Not a standard part of SQL as far as I'm aware
  - May prove useful to you at some point
  - Other DBs may have something similar

```
CREATE INDEX idx_invoice_line_amount
    ON invoice_items(UnitPrice*Quantity);
```

Indexes - Expressions

- If you are using a computed value in an expression, indexing on the result of that expression may help with specialized queries
  - Relational or sorting operations

```
SELECT InvoiceLineId,
       InvoiceId,
       UnitPrice * Quantity
FROM invoice_items
WHERE Quantity * Unitprice > 10;
```

Indexes - Expressions

- Another option here is to denormalize the column
- DBAs may hate it, but it works...

```sql
SELECT InvoiceLineId,
       InvoiceId,
       UnitPrice * Quantity
FROM invoice_items
WHERE Quantity * Unitprice > 10;
```

Triggers

- Triggers are a named bit of logic associated with a table and *triggered* on a certain event

```
SELECT InvoiceLineId,
       InvoiceId,
       UnitPrice * Quantity
FROM invoice_items
WHERE Quantity * Unitprice > 10;
```

Triggers

- Syntax

```
CREATE TRIGGER [IF NOT EXISTS]
    <trigger_name>
[BEFORE|AFTER|INSTEAD OF]
[INSERT|UPDATE|DELETE]
  ON <table_name>
  [WHEN condition]
BEGIN
 statements;
END;
```

```
CREATE TRIGGER validate_email_before_insert_employees
    BEFORE INSERT ON employees
    WHEN NEW.email NOT LIKE '%@%.%'
BEGIN
    RAISE (ABORT,'Invalid email address');
END;
```

Triggers

- Here we are verifying the format of the employees email in the database
  - Benefits to this verification approach?
  - Drawbacks?

```
CREATE TRIGGER validate_email_before_insert_employees
    BEFORE INSERT ON employees
    WHEN NEW.email NOT LIKE '%@%.%'
BEGIN
    RAISE (ABORT,'Invalid email address');
END;
```

Triggers

- Denormalizing data example
- Addresses the same issue that we used an expression index for previously

```
CREATE TRIGGER denormalize_invoice_line_total
    AFTER UPDATE
    ON invoice_items
    BEGIN
        UPDATE invoice_items
        SET TotalAmount = new.UnitPrice * new.Quantity
        WHERE InvoiceLineId = new.InvoiceLineId;
    END;
```

Triggers

- Trigger events
  - Table Related
    - BEFORE INSERT
    - AFTER INSERT
    - BEFORE UPDATE
    - AFTER UPDATE
    - BEFORE DELETE
    - AFTER DELETE
  - View Only
    - INSTEAD OF INSERT
    - INSTEAD OF DELETE
    - INSTEAD OF UPDATE

```
CREATE TRIGGER denormalize_invoice_line_total
    AFTER UPDATE
    ON invoice_items
    BEGIN
        UPDATE invoice_items
        SET TotalAmount = new.UnitPrice * new.Quantity
        WHERE InvoiceLineId = new.InvoiceLineId;
    END;
```

Triggers

- *New* and *Old* symbols
  - INSERT
    - *New* is available
  - UPDATE
    - Both *New* and *Old* are available
  - DELETE
    - *Old* is available

```
CREATE TRIGGER denormalize_invoice_line_total
    AFTER UPDATE
    ON invoice_items
    BEGIN
        UPDATE invoice_items
        SET TotalAmount = new.UnitPrice * new.Quantity
        WHERE InvoiceLineId = new.InvoiceLineId;
    END;
```

Triggers

- Advantages of triggers
  - Typically fast
  - Defined with your data model
  - Can’t be avoided via a side-channel
- Disadvantages of triggers
  - Tend to be opaque
  - Difficult to track down from a non-DB environement

```
CREATE TRIGGER denormalize_invoice_line_total
    AFTER UPDATE
    ON invoice_items
    BEGIN
        UPDATE invoice_items
        SET TotalAmount = new.UnitPrice * new.Quantity
        WHERE InvoiceLineId = new.InvoiceLineId;
    END;
```

Stored Procedures

- Many databases have a feature called *Stored Procedures*
- Stored Procedures are functions defined within the database
  - Can be *very* fast
  - Can take advantage of native database features
  - Database may heavily optimize stored procedures

```
USE AdventureWorks2012;
GO
CREATE PROCEDURE HumanResources.uspGetEmployeesTest2
    @LastName nvarchar(50),
    @FirstName nvarchar(50)
AS

    SET NOCOUNT ON;
    SELECT FirstName, LastName, Department
    FROM HumanResources.vEmployeeDepartmentHistory
    WHERE FirstName = @FirstName AND LastName = @LastName
    AND EndDate IS NULL;
GO
```

Stored Procedures

- SQLite does not support stored procedures
- I have not seen heavy use of stored procedures in industry
  - Tends to hide crucial logic in the database, away from the application

```
USE AdventureWorks2012;
GO
CREATE PROCEDURE HumanResources.uspGetEmployeesTest2
    @LastName nvarchar(50),
    @FirstName nvarchar(50)
AS

    SET NOCOUNT ON;
    SELECT FirstName, LastName, Department
    FROM HumanResources.vEmployeeDepartmentHistory
    WHERE FirstName = @FirstName AND LastName = @LastName
    AND EndDate IS NULL;
GO
```

DDL - Indexes & Triggers (& Stored Procedures)

- Indexes can be used to improve read performance
  - Key part of application performance tuning
- Indexes tend to hurt insert, update and delete speeds
  - As always: tradeoffs need to be made with empirical, realistic data
  - What is your application's data access profile?
- Triggers can be used to execute logic after events in the database
  - Loved by DBAs, hated by developers
- SQLite does not have *stored procedures*
  - A way of defining functions directly in the DB
  - Again, loved by DBAs, hated by developers


MONTANA
STATE UNIVERSITY