10
THAT®
CONFERENCE

10 THAT® CONFERENCE

THANK YOU, THAT CONFERENCE PARTNERS!

® CUNA MUTUAL GROUP

Unspecified SOFTWARE CO

NVISIA

KBX™

```
SELECT MAX(VALUE)
FROM
DATABASE.ADVANCED_FEATURES;
```

SEQUEL WRITING: THE GUIDE TO MAKING YOUR FOLLOWUP NOVEL A SUCCESS

You won't believe these 18 amazing SQL tips

About Me

Andrew Hooker

- Senior Software Engineer @ Procurated
- Algoma, WI (Outside Green Bay)
- Rails Developer
- @geekoncoffee (Github, Twitter, etc)

MY JOURNEY

- College
- Perl
- UniVerse Pick database (Not SQL)
- Oracle 10 (dev environment) / 8 (prod environment)
- SQL Server 7
- Progress 4GL / OpenEdge ABL
- Rails
- Modern Postgres (YAY)

ABOUT PROCURATED

- Connecting public sector buyers with peer-reviewed suppliers
- Hiring:
  - Software Engineers
  - ETL and SQL Developers
  - Product Manager

Disclaimers

- Won't work on everything
- Many of this isn't a good idea most of the time

BULK INSERT

```
INSERT INTO promotions (promotion_name, discount, start_date, expired_date)
VALUES
    (
        '2019 Summer Promotion',
        0.15,
        '20190601',
        '20190901'
    ),
    (
        '2019 Fall Promotion', 0.20, '20191001','20191101'
    ),
    ( '2019 Winter Promotion', 0.25, '20191201','20200101');
```

Common SQL - 1

- GROUP BY
  - Combines data for aggregate functions

```sql
SELECT product_id, sum(quantity) AS total FROM line_items
GROUP BY product_id WHERE sum(quantity) > 5
```

- Syntax Error

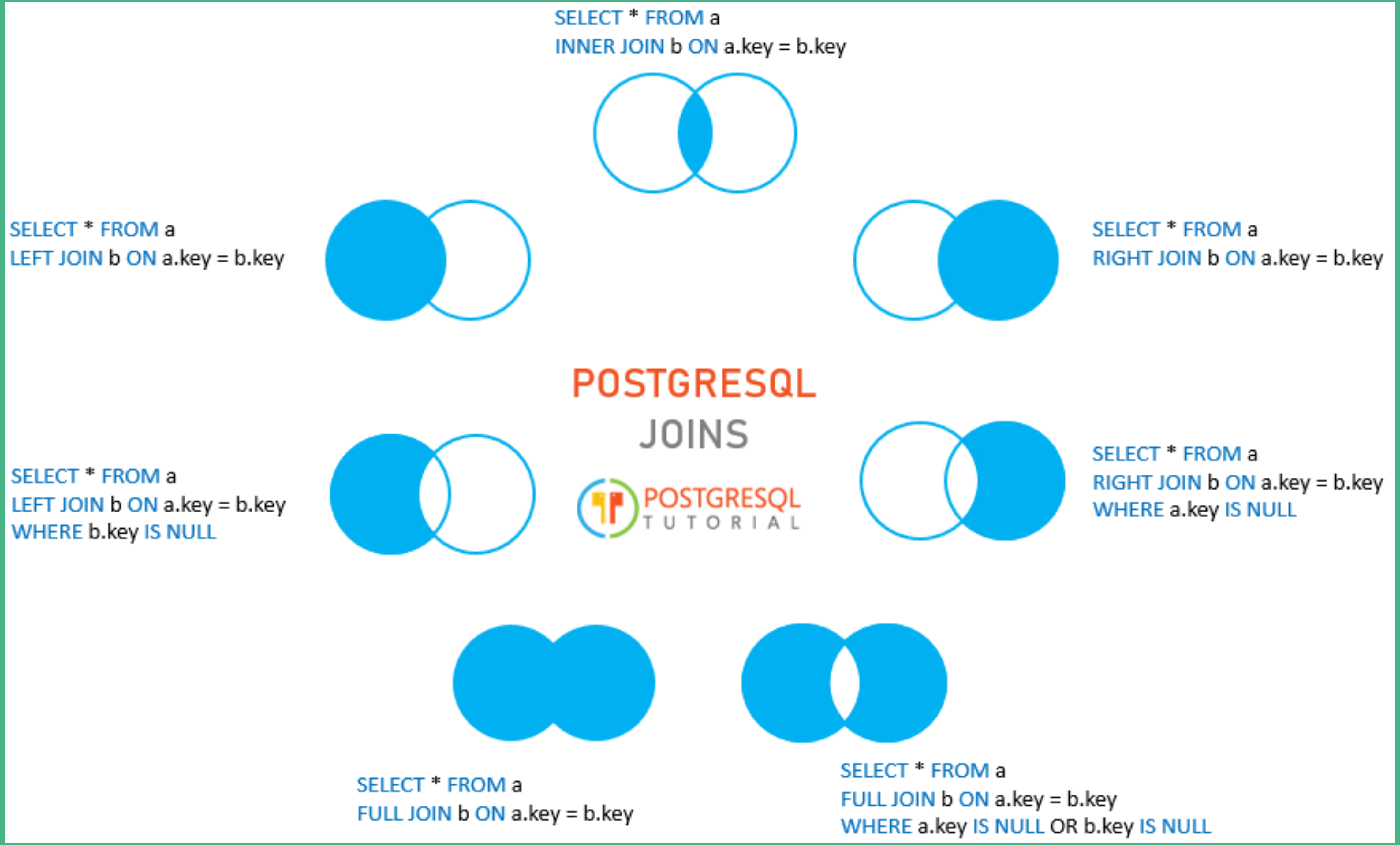
Common SQL - 2

- HAVING
  - Like 'WHERE' but for aggregates

```
SELECT product_id, sum(quantity) AS total FROM line_items
GROUP BY product_id HAVING sum(quantity) > 5
```

- works

JOINS



```
SELECT * FROM a
INNER JOIN b ON a.key = b.key
```

```
SELECT * FROM a
LEFT JOIN b ON a.key = b.key
```

```
SELECT * FROM a
RIGHT JOIN b ON a.key = b.key
```

```
SELECT * FROM a
LEFT JOIN b ON a.key = b.key
WHERE b.key IS NULL
```

```
SELECT * FROM a
RIGHT JOIN b ON a.key = b.key
WHERE a.key IS NULL
```

```
SELECT * FROM a
FULL JOIN b ON a.key = b.key
```

```
SELECT * FROM a
FULL JOIN b ON a.key = b.key
WHERE a.key IS NULL OR b.key IS NULL
```

VIEWS

- Keeping Complex Queries Organized
- Predefine complex queries
- Hyper-optimize queries to ensure performance

```sql
CREATE VIEW "Products Above Average Price" AS
SELECT id, product_name, price, featured
FROM Products
WHERE price > (SELECT AVG(price) FROM products);

SELECT * FROM "Products Above Average Price";
```

USING VIEWS

- Views can be further filtered, just like a query

```
SELECT product_name, price FROM "Products Above Average Price"
WHERE featured = true;
```

- As well as Aliased, Joined, etc

```
SELECT * FROM "Products Above Average Price" pv
LEFT OUTER JOIN products on pv.id = products.id;
```

MATERIALIZED VIEWS

- Prebuilt results of view

- Stored as a table for great performance benefit

```
CREATE MATERIALIZED VIEW top_selling_products AS
SELECT products.name, SUM(quantity) FROM line_items
LEFT OUTER JOIN products ON line_items.product_id = products.id
GROUP BY products.name
```

- Has to be manually refreshed to incorporate changes

```
REFRESH MATERIALIZED VIEW top_selling_products;
```

LATERAL JOINS / APPLY IN SQL SERVER

- Allows subqueries to reference fields from earlier SELECT FROM in the query
- Reduces the need for nested subselects

```
SELECT * FROM products JOIN LATERAL (
  SELECT
  product_id, SUM(quantity)
  FROM line_items li
  GROUP BY product_id
 ) ranked ON ranked.product_id = products.id
```

PREPARED STATEMENTS

- Allows database engine to parse and analyze a query once and run it over and over

```
PREPARE line_item_load (int, int) AS
    INSERT INTO line_items VALUES($1, $2);
EXECUTE line_item_load(1,1);
```

- Should see some performance gains in scripts which run the same query over and over again

- NOTE: only exists in a single connection, so you'll have to prepare each time

UPSERTS

- Update or Insert if Missing

```
INSERT INTO distributors AS d (id, name)
VALUES (5, 'Gizmo Transglobal'), (6, 'Associated Computing, Inc')
       ON CONFLICT (id) DO -- fields that are used for uniqueness
      UPDATE
SET name = EXCLUDED.name || ' (formerly ' || d.name || ')'
```

- EXCLUDED is a special table that contains the value you're attempting to insert

More efficient existence check

```
SELECT count(*) FROM products WHERE category_id = 1

vs

SELECT exists
  (SELECT 1
   FROM products
   WHERE category_id = 1
   LIMIT 1)
```

✳ Limit stops after it finds the first

✳ Select exists means you return 1 or 0

DIFFERENCE BETWEEN TIMES/DATES

```
SELECT
  TIMESTAMPDIFF(
    WEEK,
    '2012-09-01',
    '2014-10-01') AS NoOfWeekends1;

SELECT DATEDIFF(
  wk,
  '2012-09-01',
  '2014-10-01') AS NoOfWeekends1
```

- no postgres equivalent

Working with IP Addresses

- Why not a string?
- MySQL
  - INET(6)_ATON converts to a numeric value
  - INET(6)_NTOA converts back to an IP
- PostgreSQL
  - inet type - stores ipv4/6 addresses
- SQL Server
  - No clean option :(

Prioritize certain records in a query

```
SELECT *
FROM Users
ORDER BY IF(vip=TRUE, 0, 1),
         name
```

More control in string searching

- LIKE Matches
  - % matches 0+ characters
  - _ matches exactly 1 character

```
 SELECT * FROM products
WHERE name like '%p%'
```

- REGEX

```
 SELECT * FROM products
WHERE name ~* '.*e.*p.*'
```

COALESCE

Returns the first non-null value in a list

Helpful if a value can be stored in multiple places, for falling back to a value from a relationship, etc

```sql
SELECT COALESCE(sale_price, price) from products;
```

GENERATED COLUMNS

- Computed based on other columns, essentially a view

- Used for things like an inches version of a field stored in centimeters

- Cannot be written to, or reference anything other than the current row

```sql
CREATE TABLE people (
    height_cm numeric,
    height_in numeric GENERATED ALWAYS AS (height_cm / 2.54) STORED
);
```

Temp Tables

- Table only present in the current session
- CREATE TEMP TABLE new_tbl LIKE orig_tbl; creates a table with the structure (but not data) like the original
- Select EmployeeId,EmployeeName INTO MyTempTable from Employee Where EmployeeId>101 order by EmployeeName
- Then queryable like a regular table

Window Functions - Row Number

```
SELECT
  ROW_NUMBER() OVER (ORDER BY start_time) AS row_number,
  name
FROM
  line_items
ORDER BY created_at;
```

```
SELECT
  ROW_NUMBER() OVER ( PARTITION BY terminal
                      ORDER BY created_at),
  name
FROM
  line_items;
```

MORE WINDOW FUNCTIONS

- RANK - gives identical rows the same rank, skips # of duplicates
- DENSE_RANK - gives identical rows the same rank, then moves to next rank
- NTILE(bucket_count) - Percentile based on splitting among # of buckets
- LAG / LEAD - difference between the previous / following record

Recursive Common Table Expressions

- Get multiple recursive levels of data in a single query

```
WITH cte_org AS (
    SELECT staff_id, first_name, manager_id
      FROM sales.staffs
        WHERE manager_id IS NULL
    UNION ALL
    SELECT e.staff_id, e.first_name, e.manager_id
      FROM sales.staffs e
      INNER JOIN cte_org o
        ON o.staff_id = e.manager_id
)

SELECT * FROM cte_org;
```

ADVANCED INDEXES - 1

- B-tree
  - Postgres Default
  - Intended for data that's continually sortable
- GIN (Generalized Inverted Index)
  - Multiple keys per row
  - Arrays, json, etc

ADVANCED INDEXES - 2

- GiST (Generalized Search Tree)
  - Arbitrary splitting based on a custom attribute
  - GIS
- BRIN (Block range index)
  - Very large tables (>1M Rows)
  - (Insert/Read only tables - updates and deletes kill efficiency)
  - Data Streams / Audit Trails

Composite Key Indexes

- Not supported in all Index Types
- Index most efficient when used left to right
- Only for specialized query cases
- Not just for composite primary keys

INDEX ORDER

- ASC vs DESC
- All major engines default to ASC
- NULLS FIRST or NULLS LAST
- Pick the order which works best for your query

Stored Procedures

- Any pros want to talk about?
- Reusable Queries
- Can be secured independently from tables

Transaction Isolation Level

- Specific to database connection
- Has potentially danger side effects
- Use with Caution

Transaction Isolation Level - 1

- Read Commited
  - Generally adopted Default
  - Sees only data commited before query began
  - Sees the effects of previous updates within its transaction

Transaction Isolation Level - 2

- Read Uncommitted
  - Reads rows that have had modifications that haven't been commited
  - (Dirty Reads)
  - Avoids a lot of locks
  - Lots of potential to show things which aren't going to happen
  - Sometimes used to anticipate what's being done by long running jobs, etc

TRANSACTION ISOLATION LEVEL - 3

- Repeatable Read
    - Sees only data commited before query began
    - Ignores anything going on in the current transaction
    - More prone to failures, as data might get changed more than once in a transaction

Transaction Isolation Level - 4

- Serializable
  - Default in the SQL standard (but not in implementation)
  - Sees the latest committed data when the data is locked for access
  - Lock is held to ensure data doesn't change

Locks - 1

- Exclusive
  - Used for Insert, Update, Delete
  - Means nothing else can lock record
- Shared
  - Reserved for reading only
  - Multiple queries can issue a shared lock
  - Allows writing, but no schema changes

Locks - 2

- Update
  - Similar to Exclusive, but for a record that already has a shared lock
- Intent Locks
  - Indicates to the server the intent to acquire a lock

Locks - 3

- Schema Locks
  - Blocks access to tables while schema is being modified
- Bulk Update Locks
  - Table lock blocking other processes while a bulk import is being run

TRIGGERS

- Data Manipulation - INSERT / UPDATE / DELETE
- Data Definition - CREATE / ALTER / DROP
- User Related - LOGIN

DATA MANIPULATION TRIGGERS

- Audits
- Updating Counts
- Replication
- Many traditional uses now have better ways (stored procedures, generated columns)

Defining Data Manipulation Trigger

```
CREATE TRIGGER production.trg_product_audit ON production.products
AFTER INSERT, DELETE AS BEGIN
    SET NOCOUNT ON;
    INSERT INTO production.product_audits(product_id, list_price, updated_at, operation)
    SELECT i.product_id, i.list_price, GETDATE(), 'INS'
    FROM inserted i
    UNION ALL
    SELECT d.product_id, d.list_price, GETDATE(), 'DEL'
    FROM deleted d;
END
```

DATA DEFINITION TRIGGERS

- Logging Schema Changes

- Keeping Replication Schema in Sync

```
CREATE TRIGGER trg_index_changes
ON DATABASE
FOR
    CREATE_INDEX,
    ALTER_INDEX,
    DROP_INDEX
AS
BEGIN
    INSERT INTO index_logs (event_data, changed_by)
    VALUES (EVENTDATA(), USER);
END;
GO
```

User Triggers (SQL Server only)

- Auditing
- A Rollback in the trigger cancels the login

```
CREATE TRIGGER connection_limit_trigger
ON ALL SERVER WITH EXECUTE AS N'login_test'
FOR LOGON
AS
BEGIN
IF ORIGINAL_LOGIN()= N'login_test' AND
    (SELECT COUNT(*) FROM sys.dm_exec_sessions
            WHERE is_user_process = 1 AND
                original_login_name = N'login_test') > 3
    ROLLBACK;
END;
```

More Triggers

- INSTEAD OF - Overrides requested action
  - Could insert into an approval queue table, rather than the requested table
- SQL Server Warning - If a trigger impacts # of rows changed, unless you override (SET NOCOUNT ON), the count will update

GENERATING CSV

- Postgres

```
\COPY products TO '/Users/geekoncoffee/products.csv' DELIMITER ',' CSV HEADER;
```

- MySQL (requires FILE permission)

```
SELECT *
INTO OUTFILE '/Users/geekoncoffee/products.csv'
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
FROM products;
```

- SQL Server - No direct SQL, requires tool

NEW THINGS

- Database engines are actually remarkably stable
- Some enhancements around performance, security
- Bits and pieces enhancing datatypes, especially around JSON
- Nothing that exciting :(

Don't do it, but...

Composite Primary Keys

- Don't Do it, but...

```
CREATE TABLE orders_list (
          order_id INT,
          product_id INT,
          amount INT,
          PRIMARY KEY (order_id, product_id)
```

- Used if there's no unique identifier

- Please use an auto-increment column

CURSORS - BEGINNING

- Don't do it, but...

```
-- declare variables used in cursor
DECLARE @city_name VARCHAR(128);
DECLARE @country_name VARCHAR(128);
DECLARE @city_id INT;

-- declare cursor
DECLARE cursor_city_country CURSOR FOR
  SELECT city.id, TRIM(city.city_name), TRIM(country.country_name)
  FROM city INNER JOIN country ON city.country_id = country.id;

-- open cursor
OPEN cursor_city_country;
```

Cursors - Ending

```
-- loop through a cursor
FETCH NEXT FROM cursor_city_country INTO @city_id, @city_name, @country_name;
WHILE @@FETCH_STATUS = 0
    BEGIN
    PRINT CONCAT('city id: ', @city_id, ' / city name: ',
                 @city_name, ' / country name: ', @country_name);
    FETCH NEXT FROM cursor_city_country INTO @city_id, @city_name, @country_name;
    END;

-- close and deallocate cursor
CLOSE cursor_city_country;
DEALLOCATE cursor_city_country;
```

Build your own auto-incrementing key

- Don't do it, but...

- Anybody worked with Oracle older than 12c? (2014)

```
CREATE SEQUENCE books_sequence;
CREATE OR REPLACE TRIGGER books_on_insert
  BEFORE INSERT ON books
  FOR EACH ROW
BEGIN
  SELECT books_sequence.nextval
  INTO :new.id
  FROM dual;
END;
```

THAT®.US
JOIN US
DAILY