Click to verify

Skip to main content Powered by AI and the LinkedIn community SQL queries are the building blocks of database development, but they can also become complex, messy, and hard to understand over time. That's why documenting and commenting your SQL queries is essential for maintainability and collaboration. In this article, you'll learn how to write clear, concise, and consistent documentation and comments for your SQL queries, using some best practices and examples. M365 Architect/Developer | Microsoft Power Platform, Databases I put a comment line before a functional block and information load by my naming conventions. Also, use line breaks to separate these functional blocks. 2 We have a template for our stored procedures and have a comment block at the top to store change history. /************ --#region Header -- -- Author : Your Name -- Name : dbo.DemoProcedure -- Create Date : 26APR2023 -- Description : Description of the SP -- Example : EXECUTE dbo.DemoProcedure; -- Called By : -- -- Revision Author JIRA Description -- -- 26APR2023 Your Name JIRA TICKET Created -- --#endregion Header ************/ 1 Applications Analyst In my experience, as we re-organize our departments, documenting and commenting as we write SQL queries helps alleviate some of the stress when other developers pick up SQL queries from other teams. Comments are a way to add internal notes or annotations to SQL queries to explain or clarify the code. Headers can provide an overview of the query, such as its name, author, date, description, parameters, and output. Inline comments can explain individual lines or sections of the query, while block comments can explain larger chunks or blocks. Different syntaxes and styles may be used depending on the SQL dialect and preferences; some of the most common are single-line comments starting with -- or // and multi-line comments starting with /* and ending with */. It is important to write comments that are clear, concise, and consistent. Applications Analyst One thing we have found helpful is to have a comment box at the top of each query /*** ***/ with our overview data. Then throughout the query we use clear, concise, and consistent comments after the line of code using -- to being the comments. This allows the developer to read the query and understand it better instead of trying to remember what the overview data said 100's of lines up. To ensure your documentation and comments are effective and useful, it is important to follow certain best practices. This includes documenting and commenting your SQL queries as you write them, not after, with the intention of benefiting others. Additionally, the level of detail should be appropriate, using standard terminology and formats, as well as proper grammar, spelling, and punctuation. Additionally, consistent naming and formatting conventions should be used, rather than random or mixed ones. Documentation and comments are not static, but dynamic. To ensure they remain up-to-date and accurate, they must be reviewed and improved regularly to reflect any changes or updates in your SQL queries and database. This can involve checking for accuracy, completeness, and relevance; deleting or revising any outdated, redundant, or misleading documentation and comments; and seeking feedback from peers, clients, or users. Additionally, there are tools and techniques available to automate or simplify your documentation and comments. Applications Analyst We are implementing yearly reviews with our developer teams and our systems analysts to work together to make sure we stay up to date. This is a space to share examples, stories, or insights that don't fit into any of the previous sections. What else would you like to add? A lot of times we collaborate with our teammates to modify SQL objects like stored procedures, functions or views and release the new SQL scripts in an agile fashion through sprint cycles. In order to maintain the version history of the SQL object and to avoid re-compiling an older version of an object in the production environment, it's imperative to maintain a SQL Log on top of a stored procedure, view or any other SQL object. SQL log can be a multi-line comment that can be something like this: /* [Log] Date: Modified by: Change: */ Incrementing this log on top of a SQL object like stored procedure or function can help other team mates to check what was the last operation performed on the object and also maintain a change history. 4 Like Celebrate Support Love Insightful Funny 24 7 Contributions Documenting SQL code is essential for maintaining clarity and understanding in database development. By adding comments and descriptions within the SQL code, programmers can effectively communicate the purpose, logic, and functionality of the queries and scripts. In this guide, we will explore the importance of documenting SQL code and provide practical tips on how to effectively document your database scripts to enhance readability and facilitate collaboration among team members. Documenting SQL code is a crucial practice that every database developer and administrator should adopt. Proper documentation not only improves code maintainability but also facilitates collaboration among team members. In this guide, we will explore effective techniques on how to document SQL code in a way that is clear, concise, and beneficial for future reference. Why is SQL Code Documentation Important? Effective SQL code documentation serves numerous purposes: Enhances Readability: Well-documented code is easier to read and understand. Facilitates Collaboration: Team members can quickly grasp the functionality of SQL scripts. Aids in Troubleshooting: Clear documentation can help identify issues quickly. Supports Future Development: Code changes are easier when intent is clear. Best Practices for Documenting SQL Code 1. Use Comments Wisely Comments are the backbone of SQL code documentation. Use them effectively: Line Comments: Use -- for single line comments to explain complex logic. Block Comments: Use /…/ for multi-line comments to provide detailed explanations. Example: -- This query retrieves all orders placed in the last month SELECT * FROM orders WHERE order_date > DATEADD(month, -1, GETDATE()); 2. Write Meaningful Names for Database Objects Choosing meaningful names for tables, columns, and procedures is a key element of SQL documentation. Consider the following: Tables: Use clear, descriptive names like customer_orders instead of cryptic abbreviations. Columns: Name columns based on their content, such as order_total rather than tot. Stored Procedures: Use action-oriented names, for example, GetCustomerOrders. 3. Create a README File For larger projects, consider creating a README file that outlines the purpose, structure, and usage of your SQL code. Include: Project Overview: A brief summary of the project's functionality. Installation Instructions: How to set up the database and run the SQL scripts. Usage Guidelines: Examples of how to use the stored procedures and functions. 4. Document Your SQL Queries When writing queries, especially complex ones, provide context: /* * This query calculates the average order value per customer * for those who have placed more than 5 orders. */ SELECT customer_id, AVG(order_value) AS avg_order_value FROM orders GROUP BY customer_id HAVING COUNT(order_id) > 5; 5. Maintain Version Control Using a version control system such as Git allows you to track changes in your SQL code. Make sure to document: Change Descriptions: Explain what was altered and why. Version History: Include a version number or date in your comments. 6. Include Examples Incorporate examples within your documentation. This helps users understand how to implement your SQL queries effectively: Example: -- Example of using the GetCustomerOrders stored procedure EXEC GetCustomerOrders @CustomerId = 123; 7. Use a Consistent Formatting Style Maintain consistency in your coding style and documentation. This includes: Indentation: Use consistent indentation for readability. Capitalization: Use uppercase for SQL keywords (e.g., SELECT, FROM, WHERE). Whitespace: Employ whitespace judiciously for clarity. 8. Log All Changes and Updates It's essential to log any changes or updates to your SQL code. This log can include: Date of Change: When the modification occurred. Author: Who made the change. Description: What was changed and why. 9. Use Tools for Automated Documentation Consider leveraging tools that can help automate documentation processes: SQL Documentation Generators: Tools like DBDoc or Dataedo can automate documentation. Code Comments Plugins: Integrated development environments (IDEs) often have plugins available for enhanced commenting features. Common Mistakes to Avoid in SQL Documentation 1. Over-Commenting While comments are important, too much commentary can clutter your code. Aim for balance—comment on complex sections but avoid stating the obvious: SELECT * FROM employees; -- Selects all employees (Too obvious comment) 2. Neglecting the Alignment with Business Logic Ensure that your SQL code documentation aligns with the intended business logic. This provides context that enhances understanding for stakeholders. 3. Failing to Update Documentation Documentation must evolve alongside your code. Always update your documentation when changes are made to the SQL logic or structure. By following the best practices outlined in this guide, you can significantly enhance the quality of your SQL code documentation. Implementing thorough, meaningful, and clear documentation not only aids your current projects but also sets a standard for future developments. With the right practices in place, you will ensure that your SQL code is accessible, maintainable, and efficient for years to come. Documenting SQL code is essential for ensuring clear communication, maintaining code quality, and facilitating collaboration among team members. By following best practices and consistently annotating your code with relevant comments and explanations, you can improve readability and make it easier for others to understand and work with your SQL scripts. Taking the time to properly document your SQL code is a valuable investment that can save time and prevent errors in the long run. Data analysis involves analyzing and deriving insights from large volumes of data. SQL (Structured Query Language) is the most popular and powerful tool used to retrieve these insights (e.g., select) and manipulate data (e.g., update or delete). Businesses can make better-informed decisions by understanding the patterns, trends, and insights data analysts present. So, It is crucial to write SQL queries that are easy to read and understand for accuracy, maintainability, and efficiency.This blog post will explore ten best practices for writing SQL queries for data analysis. We will also go through some examples to make the concepts clear and easy to follow.1. Use Meaningful Alias NamesA key principle of writing clean SQL queries is using meaningful aliases for tables and columns. This practice improves query readability and helps others understand your code more easily.-- Column AliasesSELECT orders.order_id AS order_number, customers.customer_name AS customer_nameFROM ordersJOIN customers ON orders.customer_id = customers.customer_id;--Table Aliases SELECT e1.employee_name AS employee_name, e2.employee_name AS manager_nameFROM employees employeesLEFT JOIN employees managers ON employees.manager_id = managers.employee_id;2. Avoid Using SELECT *It's best to specify columns you need explicitly instead of using *, as this can result in unnecessary data transfer and slower query performance.SELECT first_name, last_name, emailFROM employees;3. Use Proper Indentation and FormattingClear and consistent indentation and formatting improve the readability and maintainability of SQL queries. Properly formatted code is easier to understand and debug.SELECT product_name, category_name, priceFROM productsWHERE price > 50;4. Avoid Subqueries When PossibleIt is recommended to use JOINs instead of subqueries for better performance as JOINs can be optimized more effectively by the database engine.SELECT customers.customer_nameFROM customersJOIN orders ON customers.customer_id = orders.customer_idWHERE orders.order_date >= '2023-01-01';5. Use Indexes, Partitioning/Bucketing for PerformanceTo enhance the speed of your query execution, it is recommended to index the columns that are used in WHERE clauses. You can also partition your data based on your access patterns, such as Order_date or event_date. Indexes can significantly improve the performance of your queries, particularly for large volumes of data.CREATE INDEX idx_customer_id ON customers(customer_id);SELECT customer_name, customer_address FROM customers WHERE customers.customer_id = 234567;6. Use Aggregate FunctionsAggregating data using functions like COUNT, SUM, and AVG is essential for extracting insights such as total order amount and number of orders placed.SELECT category_name, COUNT(*) AS product_countFROM productsGROUP BY category_name;7. Handle NULL ValuesNULL values can create problems with unexpected behavior in your SQLs. Use functions like COALESCE or NULLIF to handle NULLs effectively.SELECT product_name, COALESCE(discount, 0) AS discountFROM products;If the "discount" column contains NULL values (i.e., no discount is specified for a product), it replaces those NULL values with 0 (zero) using the COALESCE function8. Use Comments for DocumentationComments always help in understanding the logic behind any code block. Adding appropriate comments to complex or unusual queries is an excellent practice for documenting logic and helping team members understand the code for further enhancements or modifications.-- Calculate total sales for the year 2023SELECT SUM(sales_amount) AS total_salesFROM salesWHERE YEAR(sales_date) = 2023;9. Avoid Using ORDER BY in SubqueriesBoost your query performance by avoiding ORDER BY in subqueries unless it's absolutely necessary. This can help save valuable resources and improve the overall efficiency of your query.-- InefficientSELECT customer_idFROM ordersWHERE order_date = (SELECT MAX(order_date) FROM orders);-- BetterSELECT customer_idFROM ordersORDER BY order_date DESCLIMIT 1;The first SQL query uses a subquery to find the maximum order date for all the records in the "orders" table. Then, it compares this date with each order date in the main query. This process requires sorting the entire dataset in the subquery, which can be computationally expensive and utilize a lot of resources.10. Test Queries Before RunningIt is important to test your queries with a limited dataset or in a development or testing environment to avoid unintended consequences when working with live or production data.SELECT columnsFROM test_environment_db.customersWHERE condition = 'xyz';ConclusionIn order to write efficient, readable, and maintainable SQL queries, it's important to follow certain best practices. These easy and simple to follow best practices are crucial in deriving insights from data, which is the main goal of data analysis. By incorporating these practices into your workflow, you can streamline your data analysis processes and make more informed decisions. Let me know what other best practices you are using in your day-to-day data analysis work. In this article, we are going to learn some best practices that help to write more efficient SQL queries. Introduction Queries are used to communicate with the databases and perform the database operations. Such as, we use the queries to update data on a database or retrieve data from the database. Because of these functions of queries, they are used extensively by people who also interact with databases. In addition to performing accurate database operations, a query also needs to be performance, fast and readable. At least knowing some practices when we write a query will help fulfill these criteria and improve the writing of more efficient queries. Pre-requisites In this article's examples, we will use the Adventureworks2019 sample database. Deciding the appropriate SQL editor to write queries Whatever our experience in writing queries, deciding on an appropriate editor is one of the key points that will affect our productivity because, during the development of SQL queries, we mostly consume our time in the SQL editors. Preference of the SQL editor can be changed from person to person but before to decide an editor checking the features of the editor and looking at the feature comparison with its competitors will help to decide on the proper editors. In this context, Microsoft offers two different tools to develop queries: Microsoft SQL Server Management Studio (SSMS) Azure Data Studio These two tools have some advantages and disadvantages, but the main advantage of the Azure Data Studio is to can work on different platforms (Linux and macOS) and it offers a more user-friendly user interface for professionals who commonly develop queries. At the same time, the extensions allow us to add new features to it. Despite this, SQL Server Management Studio helps to manage and maintain the database administrators' operation more easily and offers a more advanced SQL query development environment. So, if we mostly consume our time developing SQL queries, using the Azure Data Studio can be more reasonable. Except then these two tools, we can also use other editors which are developed by lots of vendors. Best Practice: As much as possible as a preference to use the proper editor for your requirements and you can consider using either 3rd party add-ins or extensions that improve the capabilities of the editors. Avoid using the asterisk sign (SELECT *) in the select SQL queries Using the SELECT * statements in the queries may cause unexpected results and issues in the queries' performance. Using an asterisk sign in a query causes redundant consumption of the database engine's resources because it will retrieve all columns of the table. In particular, using SELECT * provokes consuming more network and disk resources. Another problem with using the SELECT * sign is to be facing unexpected result sets because: Column names can be changed New columns can be added The columns' order can be changed To prevent these types of problems, we need to explicitly write the column names in our SQL queries. For example, the following query will retrieve all column data of the Employee table. SELECT * FROM HumanResources.Employee However, we can transform this query correctly by explicitly defining the column names as follows and including only the columns which we needed. FROM HumanResources.Employee Using the SELECT * statement will cause performance problems. Best Practice: Besides getting rid of SELECT * statements and using the column names explicitly in the SQL queries, we can use the alias names for the tables and column names. This usage type makes our queries more readable and easily understandable. SELECT Emp.BusinessEntityID AS [Bussines Entity Id] , Emp.NationalIDNumber AS [National Number] , Emp.JobTitle AS [Job Title] , Emp.BirthDate AS [Birth Date] , Emp.MaritalStatus AS [Martial Status] , Emp.SickLeaveHours AS [Sick Leave Hours]FROM HumanResources.Employee Emp Add the requisite comments to the SQL queries SQL comments are the plain text that can be added to the queries, and they are not parsed and executed by the query engine. Mostly, we use the comments either to add some description or disable some code blocks of the queries. However, adding brief and understandable explanations to our SQL queries is one best practice because, over time, the purpose of the query and its use by the which application can be forgettable. In this case, the process of maintaining and refactoring the query will be a bit painful. Single-line comment: To change a line as a comment, we can add the two dashes (–) at the beginning of the query line, thus this line color will be changed, and these lines will not be considered by the query engine. -- This query returns the whole employee data SELECT Emp.BusinessEntityID FROM HumanResources.Employee Emp Multiple line comment: By placing multiple lines inside this sign () block, we can convert them into multiple comment lines. This query returns the whole employee data. The human resource portal uses this query SELECT p.FirstName AS [First Name] , p.LastName AS [Last Name] , e.BirthDate AS [Birth Date] FROM [HumanResources].[Employee] e INNER JOIN [Person].[Person] AS p ON p.[BusinessEntityID] = e.[BusinessEntityID] Best Practice: As possible as standardize your SQL comments and add all short information that you required according to your development process. For example, you can use the following template: --##--Purpose: Description of the business/technical details. You can use multiple lines if needed-- Jira or another number-- 01/01/0000 Developer Name full name -- A comprehensive description of the changes. The description may use as --##----------------------------------- Consider NULLable columns in SQL queries A NULL value in a row specifies an unknown value and this value does not point to a zero or empty value. Because of this particular characteristic of the NULL value, we need to take into account nullable columns in the queries. Comparing the NULL values: When we either filter out the NULL or exclude NULL rows in a query, we cannot use the equality operator (=) in the WHERE clause. The proper way to compare the NULL values is to use IS NULL and IS NOT NULL operators. For example, if we want to return only NULL values from the Address table, we can use the following query: SELECT Address.AddressLine1 AS [Adress 1] , Address.AddressLine2 AS [Adress 2]FROM Person.Address AddressWHERE Address.AddressLine2 IS NULL When we use the equality operator (=) in the WHERE clause, we do not get the appropriate result set. SELECT Address.AddressLine1 AS [Adress 1] , Address.AddressLine2 AS [Adress 2]FROM Person.Address AddressWHERE Address.AddressLine2 = NULL COUNT() function and NULLable columns: The COUNT() function counts and returns the number of columns from the query result set. However, the COUNT(*) function counts all rows of the query resultset but if we replace the column name with the asterisk sign COUNT(column_name), the function counts only the non-null values. For example when we use the asterisk (*) for the COUNT function to count the Employee table rows, we will obtain 290. SELECT COUNT(*) AS [Number of the Columns] FROM HumanResources.Employee However, if we use the column name instead of the asterisk sign the COUNT function returns a different value. SELECT COUNT(OrganizationLevel) AS [Number of the Columns] FROM HumanResources.Employee Beautify SQL Queries The queries which we write will never remain a secret and will need to be reviewed and modified by us or by other developers because they might need to test, fix or add a new feature. Because of this case, the queries we write should be as understandable and easy to read as possible. Following the suggestions below for writing more readable code will help us write more readable code. 1-Format SQL Queries: Formatting the queries is one of the important points to improve the readability of a query. A well-formatted query always significantly improves code readability. To make our queries more readable, we can take advantage of online query formatting tools or add-ins (extensions). For example, we can see the mess and complexity of the following query: DECLAre @NewValue as varchar(100)declare @I AS INT=0 if OBJECT_ID(N'tempdb..#NewTempTable')is nOT NULL BEgin dROP TABLE #NewTempTable END CREATe table #NewTempTable(Id inT,Column1 INT) IF @NewValue2aND LEN(CreditCardApprovalCode)>10ORDer by conCAT(Substring(CarrierTrackingNumber,1, 4),Substring(p.Class,1,4)),ProductID desc end SELECT Id AS [Product Id] FROM #NewTempTable Now, we will format this code through the Poor SQL format and we can see the stunning change in the query, or you can choose the formatter that suits your needs. DECLARE @NewValue AS VARCHAR(100) IF OBJECT_ID(N'tempdb..#NewTempTable') IS NOT NULL CREATE TABLE #NewTempTable (INSERT INTO #NewTempTable , s.CarrierTrackingNumber , h.CreditCardApprovalCode , dbo.[ufnGetStock](p.ProductID) AS Stock WHEN AccountNumber LIKE '10%' Substring(CarrierTrackingNumber, 1, 4) , Substring(p.Class, 1, 4) FROM Sales.SalesOrderDetailEnlarged s INNER JOIN Production.Product p ON s.ProductID = p.ProductID INNER JOIN Sales.SalesOrderHeaderEnlarged h ON h.SalesOrderID = s.SalesOrderID AND LEN(CreditCardApprovalCode) > 10 Substring(CarrierTrackingNumber, 1, 4) , Substring(p.Class, 1, 4) SELECT Id AS [Product Id] 2-Use aliases for the column and table names: We can use the alias to rename the column and table names so that we can make them more readable. For example, in the following example, the query does not use table and column allies for this reason it seems messy and difficult to read. SELECT Product.ProductID, Product.Name , WorkOrder.WorkOrderID ,WorkOrderRouting.ActualCost,WorkOrder.StockedQty from Production.WorkOrder Production.WorkOrderRouting on Production.WorkOrder.WorkOrderID = Production.WorkOrderRouting.WorkOrderID INNER JOIN Production.Product ON Production.Product.ProductID = Production.WorkOrder.ProductID Now, we will use aliases for the column names and table names to further format it. SELECT Product.ProductID AS [Product Id] , Product.Name AS [Product Name] , WorkOrder.WorkOrderID AS [WorkOrder Id] , Routing.ActualCost AS [Actual Cost] , WorkOrder.StockedQty AS [Stocked Quantity]FROM Production.WorkOrder WorkOrderINNER JOIN Production.WorkOrderRouting RoutingON WorkOrder.WorkOrderID = Routing.WorkOrderIDINNER JOIN Production.Product ProductON Product.ProductID = WorkOrder.ProductID As we can see there is a noticeable readability improvement after using and formatting the query. Best Practice: If you want to write more readable and understandable queries: Use understandable aliases for the table names Add brief comments to queries Consider using Common Table Expressions (CTE) in your complex queries Variable names should be clear and concise Format the queries before deploying Summary In this article, we learned some best practices that help to improve the quality of SQL queries. Well-formatted, more readable, and performant queries will always help us or another person who needs to review or modify the queries. 168 Views Advanced SQL Interviews don't just test you on correctness – they often check out your SQL coding style too, especially during take-home SQL challenges. Usually, they are checking to see if you follow SQL coding conventions that result in readable and maintainable SQL queries. That's why in this tutorial, we're going to teach you these 8 SQL query best practices: Uppercase for Keywords Lowercase or Snake Case for Names Descriptive and Concise Aliases Consistent Formatting and Indentation Avoid Writing SELECT * Use JOINs Explicitly for Clarity Format Dates Consistently Comment Wisely Let's start with the basic: Use uppercase for the SQL keywords and functions. It makes your queries clearer. Avoid writing it like this: Instead, opt for this format: When it comes to naming things like schemas, tables, and columns, go for lowercase or snake case (using underscores) - it's a widely adopted convention in programming languages! For example, instead of: Choose this format: Our Suggestions: While some prefer variations to distinguish schemas, tables, and columns, adhering to snake case is recommended. Remember, consistency matters, so consult your team's conventions for optimal harmony. Make your queries easy to read by using short and meaningful names for tables, columns, and expressions, and avoid the need for lengthy or ambiguous names. Avoid this approach: Instead, use this technique: SQL Writing 101: Use the "AS" keyword when assigning aliases to enhance visibility for new column or table names. For calculated columns, choose meaningful names that reflects their purpose to ensure newcomes to the query can easily understand what's being calculated. Want your queries to be easier on the eyes? Keep your formatting consistent. Use spaces or tabs throughout, and things will look neater. For example, avoid this: Instead, write like this: A little tip: You can also use a code formatter tool to automatically apply a style to your code. Don't use in your queries. Instead, explicitly list the columns you need. This improves query performance and makes the query more readable. Don't write like this: Instead, dothis: SQL Writing 101: If you're new to a table and want to see every column, add a clause to reduce the number of rows being generated: Make your queries more readable by using simple names and combining data from different tables with s. Avoid doing it like this: Instead, opt for this format: Be clear about your JOIN types (, , , etc.) to improve query clarity. Instead of being vague like this: Specify the JOIN type: Use a consistent date format to prevent ambiguity. Avoid this: Instead, use a consistent format (assuming 'YYYY-MM-DD' format): Use comments to explain your queries, but avoid writing long ones between queries. Briefly explain each step for clarity. You can use to write single-line comments in SQL. Here's an example of not-so-helpful comment: Instead, write a useful comment: A tip to remember: Use for longer comments and make sure they add valuable insights. By following these tips, your queries will be easier to understand, maintain, and work with! In the next tutorial, we'll change gears, and cover how to pivot and unpivot in SQL (similar to how PIOVT works in Excel). Have no clue what we're dtalking about? No problem! Next Tutorial: Pivoting & Un-Pivoting in SQL You can use this set of guidelines, fork them or make your own - the key here is that you pick a style and stick to it. To suggest changes or fix bugs please open an issue or pull request on GitHub. These guidelines are designed to be compatible with Joe Celko's SQL Programming Style book to make adoption for teams who have already read that book easier. This guide is a little more opinionated in some areas and in others a little more relaxed. It is certainly more succinct where Celko's book contains anecdotes and reasoning behind each rule as thoughtful prose. It is easy to include this guide in Markdown format as a part of a project's code base or reference it here for anyone on the project to freely read—much harder with a physical book. SQL style guide by Simon Holywell is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License. Based on a work at . General Do Use consistent and descriptive identifiers and names. Make judicious use of white space and indentation to make code easier to read. Store ISO 8601 compliant time and date information (YYYY-MM-DDTHH:MM:SS.SSSSS). Try to only use standard SQL functions instead of vendor-specific functions for reasons of portability. Keep code succinct and devoid of redundant SQL—such as unnecessary quoting or parentheses or WHERE clauses that can otherwise be derived. Include comments in SQL code where necessary. Use the C style opening /* and closing */ where possible otherwise precede comments with -- and finish them with a new line. SELECT file_hash -- stored ssdeep hash FROM file_system WHERE file_name = '.vimrc'; /* Updating the file record after writing to the file */ UPDATE file_system SET file_modified_date = '1980-02-22 13:19:01.00000', file_size = 209732 WHERE file_name = '.vimrc'; Avoid camelCase—it is difficult to scan quickly. Descriptive prefixes or Hungarian notation such as sp_ or tbl. Plurals—use the more natural collective term where possible instead. For example staff instead of employees or people instead of individuals. Quoted identifiers—if you must use them then stick to SQL-92 double quotes for portability (you may need to configure your SQL server to support this depending on vendor). Object-oriented design principles should not be applied to SQL or database structures. Naming conventions General Ensure the name is unique and does not exist as a reserved keyword. Keep the length to a maximum of 30 bytes—in practice this is 30 characters unless you are using a multi-byte character set. Names must begin with a letter and may not end with an underscore. Only use letters, numbers and underscores in names. Avoid the use of multiple consecutive underscores—these can be hard to read. Use underscores where you would naturally include a space in the name (first name becomes first_name). Avoid abbreviations and if you have to use them make sure they are commonly understood. SELECT first_name FROM staff; Tables Use a collective name or, less ideally, a plural form. For example (in order of preference) staff and employees. Do not prefix with tbl or any other such descriptive prefix or Hungarian notation. Never give a table the same name as one of its columns and vice versa. Avoid, where possible, concatenating two table names together to create the name of a relationship table. Rather than cars_mechanics prefer services. Columns Always use the singular name. Where possible avoid simply using id as the primary identifier for the table. Do not add a column with the same name as its table and vice versa. Always use lowercase except where it may make sense not to such as proper nouns. Aliasing or correlations Should relate in some way to the object or expression they are aliasing. As a rule of thumb the correlation name should be the first letter of each word in the object's name. If there is already a correlation with the same name then append a number. Always include the AS keyword—makes it easier to read as it is explicit. For computed data (SUM() or AVG()) use the name you would give it were it a column defined in the schema. SELECT first_name AS fn FROM staff AS s1 JOIN students AS s2 ON s2.mentor_id = s1.staff_num; SELECT SUM(s.monitor_tally) AS monitor_total FROM staff AS s; Stored procedures The name must contain a verb. Do not prefix with sp_ or any other such descriptive prefix or Hungarian notation. Uniform suffixes The following suffixes have a universal meaning ensuring the columns can be read and understood easily from SQL code. Use the correct suffix where appropriate. _id—a unique identifier such as a column that is a primary key. _status—flag value or some other status of any type such as publication_status. _total—the total or sum of a collection of values. _num—denotes the field contains any kind of number. _name—signifies a name such as first_name. _seq—contains a contiguous sequence of values. _date—denotes a column that contains the date of something. _tally—a count. _size—the size of something such as a file size or clothing. _addr—an address for the record could be physical or intangible such as ip_addr. Query syntax Reserved words Always use uppercase for the reserved keywords like SELECT and WHERE. It is best to avoid the abbreviated keywords and use the full length ones where available (prefer ABSOLUTE to ABS). Do not use database server specific keywords where an ANSI SQL keyword already exists performing the same function. This helps to make the code more portable. SELECT model_num FROM phones AS p WHERE p.release_date > '2014-09-30'; White space To make the code easier to read it is important that the correct complement of spacing is used. Do not crowd code or remove natural language spaces. Spaces Spaces should be used to line up the code so that the root keywords all end on the same character boundary. This forms a river down the middle making it easy for the readers eye to scan over the code and separate the keywords from the implementation detail. Rivers are bad in typography, but helpful here. (SELECT f.species_name, AVG(f.height) AS average_height, AVG(f.diameter) AS average_diameter FROM flora AS f WHERE f.species_name = 'Banksia' OR f.species_name = 'Sheoak' OR f.species_name = 'Wattle' GROUP BY f.species_name, f.observation_date) UNION ALL (SELECT b.species_name, AVG(b.height) AS average_height, AVG(b.diameter) AS average_diameter FROM botanic_garden_flora AS b WHERE b.species_name = 'Banksia' OR b.species_name = 'Sheoak' OR b.species_name = 'Wattle' GROUP BY b.species_name, b.observation_date); Notice that SELECT, FROM, etc. are all right aligned while the actual column names and implementation-specific details are left aligned. Although not exhaustive always include spaces: before and after equals (=) after commas (,) surrounding apostrophes (') where not within parentheses or with a trailing comma or semicolon. SELECT a.title, a.release_date, a.recording_date FROM albums AS a WHERE a.title = 'Charcoal Lane' OR a.title = 'The New Danger'; Line spacing Always include newlines/vertical space: before AND or OR after semicolons to separate queries for easier reading after each keyword definition after a comma when separating multiple columns into logical groups to separate code into related sections, which helps to ease the readability of large chunks of code. Keeping all the keywords aligned to the righthand side and the values left aligned creates a uniform gap down the middle of the query. It also makes it much easier to to quickly scan over the query definition. INSERT INTO albums (title, release_date, recording_date) VALUES ('Charcoal Lane', '1990-01-01 01:01:01.00000', '1990-01-01 01:01:01.00000'), ('The New Danger', '2008-01-01 01:01:01.00000', '1990-01-01 01:01:01.00000'); UPDATE albums SET release_date = '1990-01-01 01:01:01.00000' WHERE title = 'The New Danger'; SELECT a.title, a.release_date, a.recording_date, a.production_date -- grouped dates together FROM albums AS a WHERE a.title = 'Charcoal Lane' OR a.title = 'The New Danger'; Indentation To ensure that SQL is readable it is important that standards of indentation are followed. Joins Joins should be indented to the other side of the river and grouped with a new line where necessary. SELECT r.last_name FROM riders AS r INNER JOIN bikes AS b ON r.bike_vin_num = b.vin_num AND b.engine_tally > 2 INNER JOIN crew AS c ON r.crew_chief_last_name = c.last_name AND c.chief = 'Y'; The exception to this is when using just the JOIN keyword where it should be before the river. SELECT r.last_name FROM riders AS r JOIN bikes AS b ON r.bike_vin_num = b.vin_num Subqueries should also be aligned to the right side of the river and then laid out using the same style as any other query. Sometimes it will make sense to have the closing parenthesis on a new line at the same character position as its opening partner—this is especially true where you have nested subqueries. SELECT r.last_name, (SELECT MAX(YEAR(championship_date)) FROM champions AS c WHERE c.last_name = r.last_name AND c.confirmed = 'Y') AS last_championship_year FROM riders AS r WHERE r.last_name IN (SELECT c.last_name FROM champions AS c WHERE YEAR(championship_date) > '2008' AND c.confirmed = 'Y'); Preferred formalisms Make use of BETWEEN where possible instead of combining multiple statements with AND. Similarly use IN() instead of multiple OR clauses. Where a value needs to be interpreted before leaving the database use the CASE expression. CASE statements can be nested to form more complex logical structures. Avoid the use of UNION clauses and temporary tables where possible. If the schema can be optimised to remove the reliance on these features then it most likely should be. SELECT CASE postcode WHEN 'BN1' THEN 'Brighton' WHEN 'EH1' THEN 'Edinburgh' END AS city FROM office_locations WHERE country = 'United Kingdom' AND opening_time BETWEEN 8 AND 9 AND postcode IN ('EH1', 'BN1', 'NN1', 'KW1'); Create syntax When declaring schema information it is also important to maintain human-readable code. To facilitate this ensure that the column definitions are ordered and grouped together where it makes sense to do so. Indent column definitions by four (4) spaces within the CREATE definition. Choosing data types Where possible do not use vendor-specific data types—these are not portable and may not be available in older versions of the same vendor's software. Only use REAL or FLOAT types where it is strictly necessary for floating point mathematics otherwise prefer NUMERIC and DECIMAL at all times. Floating point rounding errors are a nuisance! Specifying default values The default value must be the same type as the column—if a column is declared a DECIMAL do not provide an INTEGER default value. Default values must follow the data type declaration and come before any NOT NULL statement. Constraints and keys Constraints and their subset, keys, are a very important component of any database definition. They can quickly become very difficult to read and reason about though so it is important that a standard set of guidelines are followed. Choosing keys Deciding the column(s) that will form the keys in the definition should be a carefully considered activity as it will effect performance and data integrity. The key should be unique to some degree. Consistency in terms of data type for the value across the schema and a lower likelihood of this changing in the future. Can the value be validated against a standard format (such as one published by ISO)? Encouraging conformity to point 2. Keeping the key as simple as possible whilst not being scared to use compound keys where necessary. It is a reasoned and considered balancing act to be performed at the definition of a database. Should requirements evolve in the future it is possible to make changes to the definitions to keep them up to date. Defining constraints Once the keys are decided it is possible to define them in the system using constraints along with field value validation. General Tables must have at least one key to be complete and useful. Constraints should be given a custom name excepting UNIQUE, PRIMARY KEY and FOREIGN KEY where the database vendor will generally supply sufficiently intelligible names automatically. Layout and order Specify the primary key first right after the CREATE TABLE statement. Constraints should be defined directly beneath the column they correspond to. Indent the constraint so that it aligns to the right of the column name. If it is a multi-column constraint then consider putting it as close to both column definitions as possible and where this is difficult as a last resort include them at the end of the CREATE TABLE definition. If it is a table-level constraint that applies to the entire table then it should also appear at the end. Use alphabetical order where ON DELETE comes before ON UPDATE. If it make senses to do so align each aspect of the query on the same character position. For example all NOT NULL definitions could start at the same character position. This is not hard and fast, but it certainly makes the code much easier to scan and read. Validation Use LIKE and SIMILAR TO constraints to ensure the integrity of strings where the format is known. Where the ultimate range of a numerical value is known it must be written as a range CHECK() to prevent incorrect values entering the database or the silent truncation of data too large to fit the column definition. In the least it should check that the value is greater than zero in most cases. CHECK() constraints should be kept in separate clauses to ease debugging. Example CREATE TABLE staff (PRIMARY KEY (staff_num), staff_num INT(5) NOT NULL, first_name VARCHAR(100) NOT NULL, pens_in_drawer INT(2) NOT NULL, CONSTRAINT pens_in_drawer_range CHECK(pens_in_drawer BETWEEN 1 AND 99)); Designs to avoid Object-oriented design principles do not effectively translate to relational database designs—avoid this pitfall. Placing the value in one column and the units in another column. The column should make the units self-evident to prevent the requirement to combine columns again later in the application. Use CHECK() to ensure valid data is inserted into the column. Entity–Attribute–Value (EAV) tables—use a specialist product intended for handling such schema-less data instead. Splitting up data that should be in one table across many tables because of arbitrary concerns such as time-based archiving or location in a multinational organisation. Later queries must then work across multiple tables with UNION rather than just simply querying one table. Appendix Reserved keyword reference A list of ANSI SQL (92, 99 and 2003), MySQL 3 to 5.x, PostgreSQL 8.1, MS SQL Server 2000, MS ODBC and Oracle 10.2 reserved keywords. A ABORT ABS ABSOLUTE ACCESS ACTION ADA ADD ADMIN AFTER AGGREGATE ALIAS ALL ALLOCATE ALSO ALTER ALWAYS ANALYSE ANALYZE AND ANY ARE ARRAY AS ASC ASENSITIVE ASSERTION ASSIGNMENT ASYMMETRIC AT ATOMIC ATTRIBUTE ATTRIBUTES AUDIT AUTHORIZATION AUTO_INCREMENT AVG AVG_ROW_LENGTH BACKUP BACKWARD BEFORE BEGIN BERNOULLI BETWEEN BIGINT BINARY BIT BIT_LENGTH BITVAR BLOB BOOL BOOLEAN BOTH BREADTH BREAK BROWSE BULK BY C CACHE CALL CALLED CARDINALITY CASCADE CASCADED CASE CAST CATALOG CATALOG_NAME CEIL CEILING CHAIN CHANGE CHAR CHAR_LENGTH CHARACTER CHARACTER_LENGTH CHARACTER_SET_CATALOG CHARACTER_SET_NAME CHARACTER_SET_SCHEMA CHARACTERISTICS CHARACTERS CHECK CHECKED CHECKPOINT CHECKSUM CLASS CLASS_ORIGIN CLOB CLOSE CLUSTER CLUSTERED COALESCE COBOL COLLATE COLLATION COLLATION_CATALOG COLLATION_NAME COLLATION_SCHEMA COLLECT COLUMN COLUMN_NAME COLUMNS COMMAND_FUNCTION COMMAND_FUNCTION_CODE COMMENT COMMIT COMMITTED COMPLETION COMPRESS COMPUTE CONDITION CONDITION_NUMBER CONNECT CONNECTION CONNECTION_NAME CONSTRAINT CONSTRAINT_CATALOG CONSTRAINT_NAME CONSTRAINT_SCHEMA CONSTRAINTS CONSTRUCTOR CONTAINS CONTAINSTABLE CONTINUE CONVERSION CONVERT COPY CORR CORRESPONDING COUNT COVAR_POP COVAR_SAMP CREATE CREATEDB CREATEROLE CREATEUSER CROSS CSV CUBE CUME_DIST CURRENT CURRENT_DATE CURRENT_DEFAULT_TRANSFORM_GROUP CURRENT_PATH CURRENT_ROLE CURRENT_TIME CURRENT_TIMESTAMP CURRENT_TRANSFORM_GROUP_FOR_TYPE CURRENT_USER CURSOR CURSOR_NAME CYCLE DATA DATABASE DATABASES DATE DATETIME DATETIME_INTERVAL_CODE DATETIME_INTERVAL_PRECISION DAY DAY_HOUR DAY_MICROSECOND DAY_MINUTE DAY_SECOND DAYOFMONTH DAYOFWEEK DAYOFYEAR DBCC DEALLOCATE DEC DECIMAL DECLARE DEFAULT DEFAULTS DEFERRABLE DEFERRED DEFINED DEFINER DEGREE DELAY_KEY_WRITE DELAYED DELETE DELIMITER DELIMITERS DENSE_RANK DENY DEPTH DEREF DERIVED DESC DESCRIBE DESCRIPTOR DESTROY DESTRUCTOR DETERMINISTIC DIAGNOSTICS DICTIONARY DISABLE DISCONNECT DISK DISPATCH DISTINCT DISTINCTROW DISTRIBUTED DIV DO DOMAIN DOUBLE DROP DUAL DUMMY DUMP DYNAMIC DYNAMIC_FUNCTION DYNAMIC_FUNCTION_CODE EACH ELEMENT ELSE ELSEIF ENABLE ENCLOSED ENCODING ENCRYPTED END END-EXEC ENUM EQUALS ERRLVL ESCAPE ESCAPED EVERY EXCEPT EXCEPTION EXCLUDE EXCLUDING EXCLUSIVE EXEC EXECUTE EXISTING EXISTS EXIT EXP EXPLAIN EXTERNAL EXTRACT FALSE FETCH FIELDS FILE FILLFACTOR FILTER FINAL FIRST FLOAT FLOAT4 FLOAT8 FLOOR FLUSH FOLLOWING FOR FORCE FOREIGN FORTRAN FORWARD FOUND FREE FREETEXT FREETEXTTABLE FREEZE FROM FULL FULLTEXT FUNCTION FUSION G GENERAL GENERATED GET GLOBAL GO GOTO GRANT GRANTED GRANTS GREATEST GROUP GROUPING HANDLER HAVING HEADER HEAP HIERARCHY HIGH_PRIORITY HOLD HOLDLOCK HOST HOSTS HOUR HOUR_MICROSECOND HOUR_MINUTE HOUR_SECOND IDENTIFIED IDENTITY IDENTITY_INSERT IDENTITYCOL IF IGNORE ILIKE IMMEDIATE IMMUTABLE IMPLEMENTATION IMPLICIT IN INCLUDE INCLUDING INCREMENT INDEX INDICATOR INFILE INFIX INHERIT INHERITS INITIAL INITIALIZE INITIALLY INNER INOUT INPUT INSENSITIVE INSERT INSERT_ID INSTANCE INSTANTIABLE INSTEAD INT INT1 INT2 INT3 INT4 INT8 INTEGER INTERSECT INTERSECTION INTERVAL INTO INVOKER IS ISAM ISNULL ISOLATION ITERATE JOIN K KEY KEY_MEMBER KEY_TYPE KEYS KILL LANCOMPILER LANGUAGE LARGE LAST LAST_INSERT_ID LATERAL LEADING LEAST LEAVE LEFT LENGTH LESS LEVEL LIKE LIMIT LINENO LINES LISTEN LN LOAD LOCAL LOCALTIME LOCALTIMESTAMP LOCATION LOCATOR LOCK LOGIN LOGS LONG LONGBLOB LONGTEXT LOOP LOW_PRIORITY LOWER M MAP MATCH MATCHED MAX MAX_ROWS MAXEXTENTS MAXVALUE MEDIUMBLOB MEDIUMINT MEDIUMTEXT MEMBER MERGE MESSAGE_LENGTH MESSAGE_OCTET_LENGTH MESSAGE_TEXT METHOD MIDDLEINT MIN MIN_ROWS MINUS MINUTE MINUTE_MICROSECOND MINUTE_SECOND MINVALUE MLSLABEL MOD MODE MODIFIES MODIFY MODULE MONTH MONTHNAME MORE MOVE MULTISET MUMPS MYISAM NAME NAMES NATIONAL NATURAL NCHAR NCLOB NESTING NEW NEXT NO NO_WRITE_TO_BINLOG NOAUDIT NOCHECK NOCOMPRESS NOCREATEDB NOCREATEROLE NOCREATEUSER NOINHERIT NOLOGIN NONCLUSTERED NONE NORMALIZE NORMALIZED NOSUPERUSER NOT NOTHING NOTIFY NOTNULL NOWAIT NULL NULLABLE NULLIF NULLS NUMBER NUMERIC OBJECT OCTET_LENGTH OCTETS OF OFF OFFLINE OFFSET OFFSETS OIDS OLD ON ONLINE ONLY OPEN OPENDATASOURCE OPENQUERY OPENROWSET OPENXML OPERATION OPERATOR OPTIMIZE OPTION OPTIONALLY OPTIONS OR ORDER ORDERING ORDINALITY OTHERS OUT OUTER OUTFILE OUTPUT OVER OVERLAPS OVERLAY OVERRIDING OWNER PACK_KEYS PAD PARAMETER PARAMETER_MODE PARAMETER_NAME PARAMETER_ORDINAL_POSITION PARAMETER_SPECIFIC_CATALOG PARAMETER_SPECIFIC_NAME PARAMETER_SPECIFIC_SCHEMA PARAMETERS PARTIAL PARTITION PASCAL PASSWORD PATH PCTFREE PERCENT PERCENT_RANK PERCENTILE_CONT PERCENTILE_DISC PLACING PLAN PLI POSITION POSTFIX POWER PRECEDING PRECISION PREFIX PREORDER PREPARE PREPARED PRESERVE PRIMARY PRINT PRIOR PRIVILEGES PROC PROCEDURAL PROCEDURE PROCESS PROCESSLIST PUBLIC PURGE QUOTE RAID0 RAISERROR RANGE RANK RAW READ READS READTEXT REAL RECHECK RECONFIGURE RECURSIVE REF REFERENCES REFERENCING REGEXP REGR_AVGX REGR_AVGY REGR_COUNT REGR_INTERCEPT REGR_R2 REGR_SLOPE REGR_SXX REGR_SXY REGR_SYY REINDEX RELATIVE RELEASE RELOAD RENAME REPEAT REPEATABLE REPLACE REPLICATION REQUIRE RESET RESIGNAL RESOURCE RESTART RESTORE RESTRICT RESULT RETURN RETURNED_CARDINALITY RETURNED_LENGTH RETURNED_OCTET_LENGTH RETURNED_SQLSTATE RETURNS REVOKE RIGHT RLIKE ROLE ROLLBACK ROLLUP ROUTINE ROUTINE_CATALOG ROUTINE_NAME ROUTINE_SCHEMA ROW ROW_COUNT ROW_NUMBER ROWCOUNT ROWGUIDCOL ROWID ROWNUM ROWS RULE SAVE SAVEPOINT SCALE SCHEMA SCHEMA_NAME SCHEMAS SCOPE SCOPE_CATALOG SCOPE_NAME SCOPE_SCHEMA SCROLL SEARCH SECOND SECOND_MICROSECOND SECTION SECURITY SELECT SELF SENSITIVE SEPARATOR SEQUENCE SERIALIZABLE SERVER_NAME SESSION SESSION_USER SET SETOF SETS SETUSER SHARE SHOW SHUTDOWN SIGNAL SIMILAR SIMPLE SIZE SMALLINT SOME SONAME SOURCE SPACE SPATIAL SPECIFIC SPECIFIC_NAME SPECIFICTYPE SQL SQL_BIG_RESULT SQL_BIG_SELECTS SQL_BIG_TABLES SQL_CALC_FOUND_ROWS SQL_LOG_OFF SQL_LOG_UPDATE SQL_LOW_PRIORITY_UPDATES SQL_SELECT_LIMIT SQL_SMALL_RESULT SQL_WARNINGS SQLCA SQLCODE SQLERROR SQLEXCEPTION SQLSTATE SQLWARNING SQRT SSL STABLE START STARTING STATE STATEMENT STATIC STATISTICS STATUS STDDEV_POP STDDEV_SAMP STDIN STDOUT STORAGE STRAIGHT_JOIN STRICT STRING STRUCTURE STYLE SUBCLASS_ORIGIN SUBLIST SUBMULTISET SUBSTRING SUCCESSFUL SUM SUPERUSER SYMMETRIC SYNONYM SYSDATE SYSID SYSTEM SYSTEM_USER TABLE TABLE_NAME TABLES TABLESAMPLE TABLESPACE TEMP TEMPLATE TEMPORARY TERMINATE TERMINATED TEXT TEXTSIZE THAN THEN TIES TIME TIMESTAMP TIMEZONE_HOUR TIMEZONE_MINUTE TINYBLOB TINYINT TINYTEXT TO TOAST TOP TOP_LEVEL_COUNT TRAILING TRAN TRANSACTION TRANSACTION_ACTIVE TRANSACTIONS_COMMITTED TRANSACTIONS_ROLLED_BACK TRANSFORM TRANSFORMS TRANSLATE TRANSLATION TREAT TRIGGER TRIGGER_CATALOG TRIGGER_NAME TRIGGER_SCHEMA TRIM TRUE TRUNCATE TRUSTED TSEQUAL TYPE UESCAPE UID UNBOUNDED UNCOMMITTED UNDER UNDO UNENCRYPTED UNION UNIQUE UNKNOWN UNLISTEN UNLOCK UNNAMED UNNEST UNSIGNED UNTIL UPDATE UPDATETEXT UPPER USAGE USE USER USER_DEFINED_TYPE_CATALOG USER_DEFINED_TYPE_CODE USER_DEFINED_TYPE_NAME USER_DEFINED_TYPE_SCHEMA USING UTC_DATE UTC_TIME UTC_TIMESTAMP VACUUM VALID VALIDATE VALIDATOR VALUE VALUES VAR_POP VAR_SAMP VARBINARY VARCHAR VARCHAR2 VARCHARACTER VARIABLE VARIABLES VARYING VERBOSE VIEW VOLATILE WAITFOR WHEN WHENEVER WHERE WHILE WIDTH_BUCKET WINDOW WITH WITHIN WITHOUT WORK WRITE WRITETEXT X509 XOR YEAR YEAR_MONTH ZEROFILL ZONE Column data types These are some suggested column data types to use for maximum compatibility between database engines. Character types: Numeric types Exact numeric types BIGINT DECIMAL DECFLOAT INTEGER NUMERIC SMALLINT Approximate numeric types DOUBLE PRECISION FLOAT REAL Binary types: Additional types Learn SQL with Metabase Download Metabase for free, or sign up for a free trial of Metabase Cloud This article covers some best practices for writing SQL queries for data analysts and data scientists. Most of our discussion will concern SQL in general, but we'll include some notes on features specific to Metabase that make writing SQL a breeze. Correctness, readability, then optimization: in that order The standard warning against premature optimization applies here. Avoid tuning your SQL query until you know your query returns the data you're looking for. And even then, only prioritize optimizing your query if it's run frequently (like powering a popular dashboard), or if the query traverses a large number of rows. In general, prioritize accuracy (does the query produce the intended results), and readability (can others easily understand and modify the code) before worrying about performance. Make your haystacks as small as possible before searching for your needles Arguably, we're already getting into optimization here, but the goal should be to tell the database to scan the minimum number of values necessary to retrieve your results. Part of SQL's beauty is its declarative nature. Instead of telling the database how to retrieve records, you need only tell the database which records you need, and the database should figure out the most efficient way to get that information. Consequently, much of the advice about improving the efficiency of queries is simply about showing people how to use the tools in SQL to articulate their needs with more precision. We'll review the general order of query execution, and include tips along the way to reduce your search space. Then we'll talk about three essential tools to add to your utility belt: INDEX, EXPLAIN, and WITH. First, get to know your data Familiarize yourself with your data before your write a single line of code by studying the metadata to make sure that a column really does contain the data you expect. The SQL editor in Metabase features a handy data reference tab (accessible via the book icon), where you can browse through the tables in your database, and view their columns and connections: You can also view sample values for specific columns: Metabase gives you many different ways to explore your data: you can X-ray tables, compose questions using the query builder, convert a saved question to SQL code, or build from an existing SQL query. We cover this in other articles; for now, let's go through the general workflow of a query. Developing your query Everyone's method will differ, but here's an example workflow to follow when developing a query. As above, study the column and table metadata. If you're using Metabase's native (SQL) editor, you can also search for Snippets that contain SQL code for the table and columns you're working with. Snippets allow you to see how other analysts have been querying the data. Or you can start a query from an existing SQL question. To get a feel for a table's values, SELECT * from the tables you're working with and LIMIT your results. Keep the LIMIT applied as you refine your columns (or add more columns via joins). Narrow down the columns to the minimal set required to answer your question. Apply any filters to those columns. If you need to aggregate data, aggregate a small number of rows and confirm that the aggregations are as you expect. Once you have a query returning the results you need, look for sections of the query to save as a Common Table Expression (CTE) to encapsulate that logic. With Metabase, you can also save code as a Snippet to share and reuse in other queries. The general order of query execution Before we get into individual tips on writing SQL code, it's important to have a sense of how databases will carry out your query. This differs from the reading order (left to right, top to bottom) you use to compose your query. Query optimizers can change the order of the following list, but this general lifecycle of a SQL query is good to keep in mind when writing SQL. We'll use the execution order to group the tips on writing good SQL that follow. The rule of thumb here is this: the earlier in this list you can eliminate data, the better. FROM (and JOIN) get(s) the tables referenced in the query. These tables represent the maximum search space specified by your query. Where possible, restrict this search space before moving forward. WHERE filters data. GROUP BY aggregates data. HAVING filters out aggregated data that doesn't meet the criteria. SELECT grabs the columns (then deduplicates rows if DISTINCT is invoked). UNION merges the selected data into a results set. ORDER BY sorts the results. And, of course, there will always be occasions where the query optimizer for your particular database will devise a different query plan, so don't get hung up on this order. Some query guidelines (not rules) The following tips are guidelines, not rules, intended to keep you out of trouble. Each database handles SQL differently, has a slightly different set of functions, and takes different approaches to optimizing queries. And that's before we even get into comparing traditional transactional databases with analytics databases that use columnar storage formats, which have vastly different performance characteristics. Help people out (including yourself three months from now) by adding comments that explain different parts of the code. The most important thing to capture here is the "why." For example, it's obvious that the code below filters out orders with ID greater than 10, but the reason it's doing that is because the first 10 orders are used for testing. SELECT id, product FROM orders -- filter out test orders WHERE order.id > 10 The catch here is that you introduce a little maintenance overhead: if you change the code, you need to make sure that the comment is still relevant and up to date. But that's a small price to pay for readable code. SQL best practices for FROM Join tables using the ON keyword Although it's possible to "join" two tables using a WHERE clause (that is, to perform an implicit join, like SELECT * FROM a,b WHERE a.foo = b.bar), you should instead prefer an explicit JOIN: SELECT o.id, o.total, p.vendor FROM orders AS o JOIN products AS p ON o.product_id = p.id Mostly for readability, as the JOIN + ON syntax distinguishes joins from WHERE clauses intended to filter the results. Alias multiple tables When querying multiple tables, use aliases, and employ those aliases in your select statement, so the database (and your reader) doesn't need to parse which column belongs to which table. Note that if you have columns with the same name across multiple tables, you will need to explicitly reference them with either the table name or alias. Avoid SELECT title, last_name, first_name FROM fiction_books LEFT JOIN fiction_authors ON fiction_books.author_id = fiction_authors.id Prefer SELECT books.title, authors.last_name, authors.first_name FROM fiction_books AS books LEFT JOIN fiction_authors AS authors ON books.author_id = authors.id This is a trivial example, but

when the number of tables and columns in your query increases, your readers won’t have to track down which column is in which table. That and your queries might break if you join a table with an ambiguous column name (e.g., both tables include a field called Created_At. Note that field filters are incompatible with table aliases, so you’ll need to remove aliases when connecting filter widgets to your Field Filters. SQL best practices for WHERE Filter with WHERE before HAVING Use a WHERE clause to filter superfluous rows, so you don’t have to compute those values in the first place. Only after removing irrelevant rows, and after aggregating those rows and grouping them, should you include a HAVING clause to filter out aggregates. Avoid functions on columns in WHERE clauses Using a function on a column in a WHERE clause can really slow down your query, as the function makes the query non-sargable (i.e., it prevents the database from using an index to speed up the query). Instead of using the index to skip to the relevant rows, the function on the column forces the database to run the function on each row of the table. And remember, the concatenation operator || is also a function, so don’t get fancy trying to concat strings to filter multiple columns. Prefer multiple conditions instead: Avoid SELECT hero, sidekick FROM superheros WHERE hero || sidekick = 'BatmanRobin' Prefer SELECT hero, sidekick FROM superheros WHERE hero = 'Batman' AND sidekick = 'Robin' Prefer = to LIKE This is not always the case. It’s good to know that LIKE compares characters, and can be paired with wildcard operators like %, whereas the = operator compares strings and numbers for exact matches. The = can take advantage of indexed columns. This isn’t the case with all databases, as LIKE can use indexes (if they exist for the field) as long as you avoid prefixing the search term with the wildcard operator, %. Which brings us to our next point: Avoid bookending wildcards in WHERE statements Using wildcards for searching can be expensive. Prefer adding wildcards to the end of strings. Prefixing a string with a wildcard can lead to a full table scan. Avoid SELECT column FROM table WHERE col LIKE "%wizar%" Prefer SELECT column FROM table WHERE col LIKE "wizar%" Prefer EXISTS to IN If you just need to verify the existence of a value in a table, prefer EXISTS to IN, as the EXISTS process exits as soon as it finds the search value, whereas IN will scan the entire table. IN should be used for finding values in lists. Similarly, prefer NOT EXISTS to NOT IN. SQL best practices for GROUP BY Order multiple groupings by descending cardinality Where possible, GROUP BY columns in order of descending cardinality. That is, group by columns with more unique values first (like IDs or phone numbers) before grouping by columns with fewer distinct values (like state or gender). SQL best practices for HAVING Only use HAVING for filtering aggregates And before HAVING, filter out values using a WHERE clause before aggregating and grouping those values. SQL best practices for SELECT SELECT columns, not stars Specify the columns you’d like to include in the results (though it’s fine to use * when first exploring tables — just remember to LIMIT your results). SQL best practices for UNION Prefer UNION All to UNION If duplicates are not an issue, UNION ALL won’t discard them, and since UNION ALL isn’t tasked with removing duplicates, the query will be more efficient. SQL best practices for ORDER BY Sorting is expensive. If you must sort, make sure your subqueries are not needlessly sorting data. SQL best practices for INDEX This section is for the database admins in the crowd (and a topic too large to fit in this article). One of the most common things folks run into when experiencing performance issues in database queries is a lack of adequate indexing. Which columns you should index usually depends on the columns you’re filtering by (i.e., which columns typically end up in your WHERE clauses). If you find that you’re always filtering by a common set of columns, you should consider indexing those columns. Adding indexes Indexing foreign key columns and frequently queried columns can significantly decrease query times. Here’s an example statement to create an index: CREATE INDEX product_title_index ON products (title) There are different types of indexes available, the most common index type uses a B-tree to speed up retrieval. Check out our article on making dashboards faster, and consult your database’s documentation on how to create an index. Use partial indexes For particularly large datasets, or lopsided datasets, where certain value ranges appear more frequently, consider creating an index with a WHERE clause to limit the number of rows indexed. Partial indexes can also be useful for date ranges as well, for example if you want to index the past week of data only. Use composite indexes For columns that typically go together in queries (such as last_name, first_name), consider creating a composite index. The syntax is similar to creating a single index. For example: CREATE INDEX full_name_index ON customers (last_name, first_name) EXPLAIN Look for bottlenecks Some databases, like PostgreSQL, offer insight into the query plan based on your SQL code. Simply prefix your code with the keywords EXPLAIN ANALYZE. You can use these commands to check your query plans and look for bottlenecks, or to compare plans from one version of your query to another to see which version is more efficient. Here’s an example query using the dvdrental sample database available for PostgreSQL. EXPLAIN ANALYZE SELECT title, release_year FROM film WHERE release_year > 2000; And the output: Seq Scan on film (cost=0.00..66.50 rows=1000 width=19) (actual time=0.008..0.311 rows=1000 loops=1) Filter: ((release_year)::integer > 2000) Planning Time: 0.062 ms Execution Time: 0.416 ms You’ll see milliseconds required for planning time, execution time, as well as the cost, rows, width, times, loops, memory usage, and more. Reading these analyses is somewhat of an art, but you can use them to identify problem areas in your queries (such as nested loops, or columns that could benefit from indexing), as you refine them. Here’s PostreSQL’s documentation on using EXPLAIN. WITH Organize your queries with Common Table Expressions (CTE) Use the WITH clause to encapsulate logic in a common table expression (CTE). Here’s an example of a query that looks for the products with the highest average revenue per unit sold in 2019, as well as max and min values. WITH product_orders AS (SELECT o.created_at AS order_date, p.title AS product_title, (o.subtotal / o.quantity) AS revenue_per_unit FROM orders AS o LEFT JOIN products AS p ON o.product_id = p.id -- Filter out orders placed by customer service for charging customers WHERE o.quantity > 0) SELECT product_title AS product, AVG(revenue_per_unit) AS avg_revenue_per_unit, MAX(revenue_per_unit) AS max_revenue_per_unit, MIN(revenue_per_unit) AS min_revenue_per_unit FROM product_orders WHERE order_date BETWEEN '2019-01-01' AND '2019-12-31' GROUP BY product ORDER BY avg_revenue_per_unit DESC The WITH clause makes the code readable, as the main query (what you’re actually looking for) isn’t interrupted by a long sub query. You can also use CTEs to make your SQL more readable if, for example, your database has fields that are awkwardly named, or that require a little bit of data munging to get the useful data. For example, CTEs can be useful when working with JSON fields. Here’s an example of extracting and converting fields from a JSON blob of user events. WITH source_data AS (SELECT events->'data'->>'name' AS event_name, CAST(events->'data'->>'ts' AS timestamp) AS event_timestamp CAST(events->'data'->>'cust_id' AS int) AS customer_id FROM user_activity) SELECT event_name, event_timestamp, customer_id FROM source_data Alternatively, you could save a subquery as a Snippet: And yes, as you might expect, the Aerodynamic Leather Toucan fetches the highest average revenue per unit sold. SQL is amazing. But so is Metabase’s Query Builder. You can compose queries using Metabase’s graphical interface to join tables, filter and summarize data, create custom columns, and more. And with custom expressions, you can handle the vast majority of analytical use cases, without ever needing to reach for SQL. Questions composed using the Query Builderr also benefit from automatic drill-through, which allows viewers of your charts to click through and explore the data, a feature not available to questions written in SQL. Glaring errors or omissions? There are libraries of books on SQL, so we’re only scratching the surface here. You can share the secrets of your SQL sorcery with other Metabase users on our forum. CTEs are named sets of results that help keep your code organized. They allow you to reuse results in the same query, and perform multi-level aggregations. Next article