# How to prepare a database for CodeQL

* CodeQL treats code like data and helps identify security vulnerabilities and errors.
* To analyze code using CodeQL, you need to create a database containing the code's relational data.
* The CodeQL CLI is used to analyze code and generate a database representation.
* You need to install and set up the CodeQL CLI and check out the version of your codebase to be analyzed.
* For compiled languages, the code directory should be ready to build with dependencies installed.
* For interpreted languages, the extractor runs directly on the source code, capturing an accurate representation.
* Set up the CodeQL CLI by downloading the zip package and creating a CodeQL directory.
* Obtain a local copy of the CodeQL queries repository and extract the zip archive.
* Launch the CodeQL CLI using the extracted files or by adding it to your PATH.
* Verify the setup by executing CodeQL CLI subcommands to ensure languages and QL packs are available.
* Create a CodeQL database using the CLI command: "codeql database create <database> --language=<language-identifier>"
* Understand that databases contain relational data and have a specific schema for each language.
* CodeQL databases have expressions and statements tables, providing an abstraction layer for analysis.
* Consider potential shortfalls in database creation, such as using a language matrix and autobuild behavior.
* Optionally, use the CodeQL extension in Visual Studio Code for compiling and running queries.

# Run CodeQL in a database

* After extracting code to a database, you can analyze it using CodeQL queries.
* CodeQL queries are written by GitHub experts, security researchers, and community contributors.
* You can also write your own custom queries to identify problems and security vulnerabilities in your code.
* There are two important types of queries: alert queries and path queries.
* Alert queries highlight issues in specific locations of your code, while path queries describe the flow of information between a source and a sink in your code.
* CodeQL queries have metadata properties that provide information about their purpose and how to interpret and display the results.
* Queries are written in QL, a programming language similar to SQL, and support logical formulas, predicates, recursion, and aggregates.
* Path queries are used to visualize the flow of information through a codebase and require defining path graph modules, source and sink nodes, and message explanations.
* To analyze a CodeQL database, you can use the codeql database analyze command and obtain results in Static Analysis Results Interchange Format (SARIF) or another interpreted format.
* SARIF files can be used to share static analysis results and can have categories to distinguish different analyses.
* SARIF results can be uploaded to GitHub using the codeql github upload-results command, requiring authentication with a GitHub App or personal access token.
* Third-party SARIF results can also be uploaded, but they need to follow the specific SARIF 2.1.0 JSON schema and include fingerprint data to avoid duplicate alerts in code scanning.

# Understand CodeQL results

* CodeQL analysis results can be viewed and analyzed to identify security vulnerabilities.
* Interpreted query results are displayed in the source code within CodeQL for Visual Studio Code.
* Results generated by the CodeQL CLI can be output in various formats for use with different tools.
* The select statement in a query can be modified to control how analysis results are displayed in the source code.
* Code scanning alerts are set up to check the code in a repository, including default CodeQL analysis, third-party analysis, or custom queries.
* Alerts from different sources are displayed together in the repository, with GitHub's default CodeQL analysis providing more properties.
* Each alert includes information about the identified problem, the triggering line of code, severity, security severity, introduction time, and nature of the problem.
* CodeQL analysis provides information on how to fix the identified problems, and it can also detect data-flow problems in the code.
* Data-flow analysis in code scanning identifies potential security issues related to insecure data usage, dangerous function arguments, and sensitive information leakage.
* Data-flow alerts in code scanning show how data moves through the code, helping identify areas that leak sensitive information and potential entry points for malicious attacks.
* Code scanning alerts have severity levels: Error, Warning, and Note. Errors result in check failures by default.
* Security severity levels for code scanning alerts are determined using the Common Vulnerability Scoring System (CVSS) and can be Critical, High, Medium, or Low. Critical and High severity alerts cause check failures by default.
* Alerts can be closed by fixing the problem in the code or by dismissing the alert.
* Dismissing an alert indicates that it doesn't need to be fixed or that the effort to fix it outweighs the potential benefit.
* Alerts can be dismissed from code scanning annotations in the code or from the summary list in the Security tab.
* Dismissing an alert affects all branches, removes the alert from the current alerts count, moves it to the "Closed" list, records the reason for closure, and prevents the same code from generating the alert in future code scanning runs.
* Admin permissions for the repository allow you to delete code scanning alerts.
* Deleting alerts is useful when removing a code scanning tool or reducing the set of queries used in CodeQL analysis.
* Alerts can be deleted from the summary list in the Security tab.
* When an alert is deleted:
  + It is deleted across all branches.
  + The number of current alerts for the project is reduced.
  + It is not added to the "Closed" list in the summary of alerts.
  + If the code that generated the alert remains unchanged and the same code scanning tool is run again without configuration changes, the alert will reappear in the analysis results.

# Troubleshoot CodeQL results

Section 1: Optimize CodeQL analysis runtimes

* Increase memory or the number of cores for self-hosted runners.
* Use a matrix in your workflow to speed up analysis of multiple languages.
* Reduce code size by excluding test code or analyzing subsets with each scan.
* Consider triggering analysis only on schedule events if it's too slow during push or pull\_request events.

Section 2: Analysis still failing on the default branch

* Check if Dependabot authored the commit.
* Verify if the pull request with the commit was merged using "@dependabot squash and merge."
* Avoid using the Dependabot command if code scanning and Dependabot security/version updates are enabled.
* Enable auto-merge for your repository to automatically merge pull requests.

Section 3: Best practices for optimizing CodeQL queries

* Eliminate Cartesian products by properly relating variables in predicates.
* Avoid using unrelated or negated variables that result in a large table of results.
* Use specific types in predicates to provide an upper bound on relation size.
* Using more specific types helps the query optimizer remove or specialize redundant parts.

Section 1: Determine the most specific types of a variable

* Use getAQlClass() to find the most specific types of an entity in CodeQL.
* It returns a list of types for each entity, potentially resulting in a large table of results.
* Useful for debugging but should be excluded from the final query for performance reasons.

Section 2: Avoid complex recursion

* Keep recursive predicates simple and define a base case along with a single recursive call.
* The query optimizer can remove redundant parts and optimize transitive closures for faster computation.

Section 3: Fold predicates

* Splitting predicates into smaller, linear chunks with tight bindings can assist the query optimizer.
* Use principles like preventing excessive branching and maximizing variable connections for optimization.

Section 1: Debugging data-flow queries using partial flow

* Use partial flow to debug data-flow queries that don't produce expected results.
* Data flow analysis computes possible variable values at different program points.
* Example query provided with steps for debugging data-flow queries.

Section 2: Debug artifacts for CodeQL

* Obtain debug artifacts to assist in troubleshooting CodeQL.
* Debug artifacts include CodeQL logs, databases, and SARIF files.
* Enable debug mode in the CodeQL workflow file to generate debug artifacts.

Section 3: Troubleshooting CodeQL for Visual Studio Code

* Visual Studio Code extension log files provide detailed information for troubleshooting.
* Progress and error messages are displayed as notifications in the workspace.
* Select the appropriate logs from the Output view to access the necessary information.

Section 4: Common error messages

* Familiarize yourself with common error messages for troubleshooting CodeQL workflows.
* Examples of error messages include "Server error" and "Out of disk" or "Out of memory".
* Appropriate actions to take when encountering each error message.

Section 1: Error: 403 "Resource not accessible by integration" when using Dependabot

* Dependabot is considered untrusted and runs with read-only scopes.
* Use the pull\_request event instead of push event for Dependabot branches.
* Example configuration for running workflows on specific branches.

Section 2: Warning: "git checkout HEAD^2 is no longer necessary"

* Old CodeQL workflows may show a warning about unnecessary git checkout.
* Remove the specified lines from the CodeQL workflow to address the warning.
* Example of fixing the issue by modifying the workflow steps.

Section 3: Important points about CodeQL and the QL language

* CodeQL predicates and classes are represented as database tables.
* Large predicates generate expensive computations due to large tables.
* QL language uses standard database operations like join and projection.
* Queries are evaluated bottom-up based on predicate dependencies.

Section 4: Analysis of Python code

* CodeQL analysis may produce different results on different platforms.
* Analyzing Python code can yield more results on GitHub-hosted runners with Linux.
* To disable auto-installation of Python dependencies, modify the "Initialize CodeQL" step in the workflow.

Remember:

1. Dependabot: Use pull\_request event for Dependabot branches.
2. Warning: Remove unnecessary git checkout from CodeQL workflow.
3. CodeQL: Predicates as tables, expensive computations for large predicates.
4. Python Analysis: Platform differences and disabling auto-installation.