ChatGPT

Boost.SQLite – Domain Context and Technical Deep Dive

Why SQLite C++ Wrappers Exist

The Problem: SQLite's official interface is a C API. It requires manual management of resources (opening/closing connections, finalizing statements) and explicit error checking via return codes. By default, "all SQL must be converted into a prepared statement before it can be run" [1] and each prepared statement must be explicitly finalized to avoid memory leaks. In C, this means developers must remember to call functions like `sqlite3_finalize()` for every `sqlite3_stmt` and handle error codes after every call, which is error-prone. C++ wrappers arose to make SQLite integration **safer and more convenient** for C++ developers by leveraging RAII (Resource Acquisition Is Initialization) for resource management, exceptions or robust error handling mechanisms, and type-safe abstractions.

RAII and Resource Safety: A core benefit of C++ wrappers is automatic resource cleanup. For example, the popular SQLiteCpp library notes it *"is designed using the RAII idiom"* so that each database or statement object "is always valid until destroyed" [2]. In practice, this means database connections and prepared statements are closed/finalized in object destructors, ensuring no resource leaks even if an error occurs. RAII wrappers also often provide transaction guards (see below) that automatically rollback uncommitted transactions on scope exit. This relieves developers from manually pairing every `BEGIN` with a `COMMIT` or `ROLLBACK` in error conditions. Overall, RAII makes using SQLite in C++ safer by tying the database resources' lifetimes to C++ object lifetimes.

Error Handling Improvements: The SQLite C API indicates errors via integer codes (e.g., return values of functions or `sqlite3_errmsg`) which programmers must check. C++ wrappers typically offer more robust error handling. Many use exceptions to propagate SQLite errors: for instance, SQLiteC++ will throw on any SQLite error rather than requiring manual checks [2]. This leads to cleaner code, as error paths don't have to be handled immediately after every call. Some wrappers (including Boost.SQLite) go further by providing both throwing and non-throwing interfaces. This dual approach lets developers choose exceptions for convenience or error-code methods for finer control in performance-critical or embedded scenarios. (Boost.SQLite's approach is detailed in the deep dive section.)

Type Mapping and Safety: SQLite's type system is dynamic – each value has an associated type tag at runtime, and columns can store different types at different times. The C API represents data through types like `sqlite3_value` and requires the user to retrieve values with the correct `sqlite3_column_*` function (e.g., `sqlite3_column_int`, `sqlite3_column_text`) based on the expected type. C++ wrappers improve this by mapping SQLite types to C++ types in a safer way. They often provide variant-like containers or conversion routines so that, for example, a SQLite `TEXT` or `INTEGER` value can be obtained directly as a `std::string` or `int` without error-prone casts. In fact, *"SQLite has its own opaque internal variant (called `sqlite3_value`)"* which a wrapper can encapsulate [3]. Modern wrappers like Boost.SQLite leverage C++17 features such as `std::variant` (or Boost.Variant2) to represent **"variants"** of the fundamental SQLite types (NULL, integer, text, etc.) in a type-safe union [4]. This means a query

result or parameter can be handled as a single C++ object that holds one of multiple types, instead of juggling raw pointers and type codes.

Prepared Statements & Binding: Rather than constructing SQL strings with values embedded (which risks SQL injection and wastes parsing effort), SQLite uses *prepared statements* with placeholders. A **prepared statement** is compiled SQL that can be executed multiple times with different inputs. It is essentially *"the compiled object code"* for an SQL query [5]. Wrappers make using prepared statements more ergonomic. They provide C++ classes for prepared statements and binder interfaces to bind parameters by index or name. According to SQLite's docs, *"SQLite allows SQL statements to contain parameters which are 'bound' to values prior to being evaluated"* [6]. In raw C this requires multiple function calls (`sqlite3_bind_int` etc.), but a wrapper might let you simply pass a tuple or use an `operator<<` to bind values. By abstracting this, wrappers encourage the safe practice of using bound parameters (avoiding string concatenation for SQL) and manage the lifecycle of the `sqlite3_stmt` (finalizing it automatically via RAII).

Transactions: SQLite transactions ensure that a group of operations either all succeed or all fail together (atomicity). In SQLite, *"no reads or writes occur except within a transaction"* [7], and transactions can be managed via `BEGIN/COMMIT/ROLLBACK` SQL or the C API. Wrappers help by offering transaction objects/guards. These start a transaction (usually by executing `BEGIN`) on construction and will automatically roll back if the scope exits before an explicit commit. This pattern prevents mistakes like forgetting to handle an error and leaving a transaction open. It also simplifies code: instead of manually issuing a BEGIN and ensuring every code path either commits or rolls back, a RAII **Transaction** object will do this housekeeping. The developer simply creates the object (which begins the transaction) and calls a commit method when ready; if no commit happens, the destructor rolls back. This greatly reduces boilerplate and ensures the SQLite database isn't left mid-transaction due to an exception or early return.

Custom Functions (Scalar/Aggregate): SQLite allows applications to define new SQL functions in C. These can be scalar (return a single value for a given input row) or aggregate (accumulate state over multiple rows and return a final value). The C API for this is `sqlite3_create_function()`, which *"creates new SQL functions – either scalar or aggregate"* [8], accepting function pointers for the implementation. C++ wrappers exist to make defining these functions easier and type-safe. Rather than requiring a static C callback with a `sqlite3_context*` and `sqlite3_value**` parameters, a wrapper can let you register a C++ lambda or functor directly. Internally it will bridge to the C API. This means you can write a normal C++ function and have it callable from SQL. Wrappers may also provide higher-level abstractions for aggregate functions – for example, Boost.SQLite lets you register an object with a `step()` and `final()` member, as seen later. In short, wrappers simplify extending SQLite with custom computation by hiding the details of `sqlite3_create_function` and the context handling, so developers can use idiomatic C++ for defining new SQL functions.

Virtual Tables: *Virtual tables* are an advanced SQLite extension mechanism allowing a database table's storage and behavior to be implemented in custom code (C/C++). They enable use cases like querying data in an external format (CSV, network service, custom memory structure) using SQL. The C API involves implementing the `sqlite3_module` interface and registering it via `sqlite3_create_module()` [9]. This is complex, requiring multiple callback pointers for query planning, cursor management, etc. C++ wrappers aim to lower this difficulty by offering classes or macros to define virtual tables in a more structured way. A wrapper might provide base classes for a virtual table and cursor, where you override methods instead of filling out a C struct. This turns the task into writing C++ subclasses rather than dealing

directly with C function pointers. In essence, wrappers let developers create custom virtual tables (and thereby table-valued functions) with less boilerplate, making SQLite's powerful extensibility more accessible.

Hooks and Callbacks: SQLite provides several callback hooks for events like commits, rollbacks, updates, etc. For instance, using the C API one can register an update hook with `sqlite3_update_hook()` to be notified whenever a row is changed in the database. The official documentation states: *"the sqlite3_update_hook() interface registers a callback function... to be invoked whenever a row is updated, inserted or deleted"* [10]. Similarly, `sqlite3_commit_hook()` and `sqlite3_rollback_hook()` let you intercept transaction commit or rollback events. These are powerful but again require C function pointers and global/static handling if used directly. C++ wrappers typically wrap these in cleaner interfaces – e.g., taking a `std::function` or lambda to call on such events. This allows C++ programs to react to database changes (for logging, caches, etc.) without dropping down to C. Wrappers also ensure that the hook remains in scope or appropriately disconnected when the database connection closes, preventing callback on dead objects. In summary, wrappers provide *"event hooks"* in a C++ friendly way, integrating SQLite's callback mechanisms with modern C++ constructs.

JSON and Variant Mapping: With the rise of JSON data, recent versions of SQLite include a built-in JSON extension (JSON1). *"SQLite has built-in support for JSON"* (JSON functions are included by default in modern builds) [11], treating JSON text as a data type that can be queried with special functions and operators. However, working with JSON text via the C API means manually parsing or constructing JSON strings. C++ wrappers can ease this by integrating with JSON libraries. For example, a wrapper might let you bind a JSON value (like a `boost::json::value` or `nlohmann::json`) directly to a query parameter, or retrieve a JSON column as a C++ JSON object. Boost.SQLite specifically advertises *"json support"* [12] – it can interface with **Boost.JSON** to map JSON text to a `boost::json::value` automatically. Likewise, because SQLite's dynamic typing can be viewed through a variant lens, Boost.SQLite uses a `boost::variant2` internally to handle parameter and result values [13]. This means the library can deliver query results as a `boost::variant` of possible types (int, double, string, etc.), or accept a variant when binding parameters, simplifying code that deals with unknown or mixed types. In short, wrappers exist to bridge SQLite's untyped nature with C++'s type-rich world – JSON support and variants are modern features that make it easier to work with complex or dynamic data stored in SQLite.

Summary: SQLite C++ wrappers exist to make the ubiquitous SQLite library more *"comfortable and safe to integrate"* in C++ projects. They handle the low-level details of the C API – resource cleanup, error codes, type conversions, and extension interfaces – and expose them as higher-level C++ constructs (classes, exceptions, overloaded operators, templates). By doing so, wrappers let C++ developers focus on SQL logic and application code rather than C boilerplate. As the author of Boost.SQLite describes, *"sqlite provides an excellent C-API, so this library does not attempt to hide, but to augment it."* [14] Wrappers build on SQLite's solid foundation, adding things like RAII, type safety, and extensibility in a way that feels natural in modern C++. In the next section, we delve into Boost.SQLite specifically, examining how it implements these ideas and what features it brings as a candidate Boost library.

Boost.SQLite Deep Dive (Features and Design)

Standout Features: Boost.SQLite is a thin C++17 wrapper around SQLite with some notable extensions beyond basic usage. In comparison to existing wrappers, the author notes that it *"provides more functionality when it comes to hooks, custom functions & virtual tables"* and even offers a **non-throwing** error-handling mode, plus integration with Boost's JSON and variant types [4]. These capabilities make Boost.SQLite stand

out as a comprehensive SQLite wrapper that not only covers simple CRUD operations safely but also allows advanced SQLite customization from C++.

Error Model – Exceptions and Non-Throwing APIs

Boost.SQLite's error handling is flexible, providing two forms of every operation: one that throws on failure, and one that returns an error code. By default, if you call a Boost.SQLite function without passing an error object, it will use exceptions. Specifically, any failure in those overloads will throw a `boost::system::system_error` (which encapsulates an `sqlite3` error code and message) [15]. For example, `conn.execute(sql)` or `sqlite::backup(source, target)` without error parameters will throw on error. However, nearly every operation also has an overload that accepts a `boost::system::error_code&` (and often an `error_info&` for extended messages) as the last parameters. When you use those, the function will not throw but instead set the error code (`ec`) if something goes wrong. This design is evident in the function signatures: e.g., there is:

- `auto boost::sqlite::create_module(connection& con, cstring_ref name, T&& module)` **which throws on error**, and
- `auto boost::sqlite::create_module(connection& con, cstring_ref name, T&& module, system::error_code &ec, error_info &ei)` **which returns** (typically a reference to the module object) **and reports errors via** `ec` **and** `ei` [16].

The library's documentation explicitly highlights this dual approach. It states that when using the throwing overload, "`system_error` *[is thrown] when the overload without* `ec` *is used.*" [17] Conversely, if you choose the overload with `system::error_code`, no exception is thrown; instead you check the `ec` value. The additional `boost::sqlite::error_info` provides the detailed error message from SQLite (since `sqlite3_errmsg` can offer more context). This error model is similar to other Boost libraries (like Boost.Asio) and gives developers control: in performance-sensitive code or in environments where exceptions are undesirable, one can use the non-throwing forms. In more straightforward scenarios, using the throwing form leads to cleaner code. Internally, Boost.SQLite defines an `boost::sqlite::error` category for SQLite's error codes and uses it to construct the exceptions [18], so that `system_error`'s `code()` can be examined (e.g., to distinguish constraint violations vs. IO errors, etc.).

Identifiers: Key types and functions for error handling in Boost.SQLite include the exception type `boost::sqlite::error` (which holds an error code and message) [19], the extended info struct `boost::sqlite::error_info` [20], and the use of `boost::system::error_code` in many function overloads. The design pattern is consistent: for instance, `sqlite::backup` has both `backup(source, target, ec, ei)` returning `void` and `backup(source, target)` which throws on failure [21]. Similarly, registering a custom function or module has both forms, with the docs indicating *"Exceptions:* `system::system_error` *when the overload without* `ec` *is used."* [22]. This dual API is a deliberate design choice to accommodate both Boost's audience that often uses exceptions and those who prefer error codes (e.g., in embedded systems or real-time applications).

Transactions – RAII Guards and Savepoints

Managing transactions is a strong point of Boost.SQLite. It provides an easy RAII mechanism to ensure transactions are properly closed. The class `**boost::sqlite::transaction**` is described as *"a simple transaction guard implementing RAII for transactions."* [23] Using it is straightforward: constructing a

`boost::sqlite::transaction` object on a connection will `BEGIN` a new transaction, and when that object is destroyed, if it has not been committed, it will automatically roll back the transaction. The library likely uses SQLite's `sqlite3_exec("BEGIN")` and `sqlite3_exec("ROLLBACK")` under the hood for this. To commit, the transaction object provides a `commit()` member function; calling it will commit the transaction (via `COMMIT`) and mark the guard as finished so that it won't rollback on destruction. This pattern prevents common errors like forgetting to handle an exception in the middle of a transaction – with RAII, any exception will unwind the stack, and the transaction's destructor will safely rollback, preserving database consistency.

For nested or partial transactions, Boost.SQLite also offers `**boost::sqlite::savepoint**` with similar RAII semantics [24]. SQLite's native mechanism for nested transactions is the SAVEPOINT feature, and this class wraps that. You can create a savepoint (which corresponds to `SAVEPOINT name` SQL command) and later rollback to it or release it. The `boost::sqlite::savepoint` guard will release (commit) the savepoint on destruction if it wasn't rolled back manually, or roll it back if the scope exits prematurely. Notably, savepoints can be used recursively (they can nest), and the documentation confirms *"Savepoints can be used recursively."* [25]. The presence of both `transaction` and `savepoint` types means Boost.SQLite supports both one-level transactions and finer-grained subtransactions in a nested way, all with RAII safety.

Usage Example: Using these is as simple as:

```
boost::sqlite::connection conn("database.db");
{
    boost::sqlite::transaction txn(conn);
    // perform multiple SQL statements via conn
    // ...
    txn.commit();
} // if commit not called, ~transaction will rollback
```

This pattern ensures that any early return or exception in the transaction scope triggers a rollback. Similarly, one could create `boost::sqlite::savepoint sp(conn, "SPName");` for a named savepoint (the interface likely chooses a name automatically if not provided). The key identifiers here are the classes `boost::sqlite::transaction` and `boost::sqlite::savepoint` themselves, which Boost.SQLite provides for safe transaction handling.

Extensibility – Custom Functions, Virtual Tables, and Hooks

One of Boost.SQLite's major strengths is exposing SQLite's extension points in idiomatic C++ ways:

- **User-Defined Functions:** Boost.SQLite allows defining both scalar and aggregate SQL functions using C++ callables. The library provides free functions `boost::sqlite::create_scalar_function` and `boost::sqlite::create_aggregate_function` to register new functions on a connection [26] [27]. These are templates that take the C++ functor (or lambda) implementing the function. For a *scalar function*, the functor can be something like `Func(context, args...) -> result`. For an

aggregate function, Boost.SQLite expects the functor to be an object with two specific member functions: a `step` function and a `final` function. The documentation specifies that "`func` *needs to be an object with two functions:* `void step(State&, span<sqlite::value, N> args);` *and* `T final(State&)`." When an aggregate is used in a query, Boost.SQLite will create a fresh `State` (which can be any type, often the functor itself contains the state) for each group, call `step` repeatedly for each row in the group, then call `final` at the end to get the result [28]. This design cleverly uses C++ overload resolution and templates to deduce `N` (the number of arguments) and `State` type. The result is that a user can define an aggregate by simply writing a struct with those two methods. For example, if you want a custom aggregate `collect_names` that concatenates text values, you can write a struct with a `std::string` accumulating member, a `step(std::string& accum, span<sqlite::value,1> args)` that appends the new text, and a `final(std::string& accum)` that returns the accumulated string. Passing an instance of this struct to `create_aggregate_function(conn, "collect_names", CollectNames{})` registers it. Internally, the wrapper handles calling your `step`/`final` via the C API's callbacks. This is far more convenient than using `sqlite3_create_function` directly with its C callback signature. The library's approach provides type safety (e.g., you can use `args[0].get_text()` which returns a string_view for a TEXT argument [29]) and automatically manages the function context.

- **Virtual Tables:** To support virtual tables, Boost.SQLite offers a mechanism to register a custom virtual table module using C++ classes. The primary interface is the function `boost::sqlite::create_module(connection&, name, module_object)`, which registers a virtual table module under the given name [30]. The `module_object` is a user-provided object that implements the required virtual table interface. Boost.SQLite's documentation and code provide a framework (likely through base classes in `boost::sqlite::vtab` namespace) for defining that module. For example, there are classes like `boost::sqlite::vtab::cursor` and tags like `boost::sqlite::vtab::transaction`, `vtab::modifiable`, etc., which correspond to capabilities of the virtual table [31] [32]. The `BOOST_SQLITE_EXTENSION(Name, Conn)` macro [33] is provided to declare a new module more easily. Under the hood, this ties into SQLite's `sqlite3_module` system. The key takeaway is that Boost.SQLite makes creating a virtual table less daunting by providing C++ abstractions: you implement a few member functions (for querying, updating, etc.) in a class, and the library registers it as a virtual table. As a result, something quite difficult to do in pure C (involving a lot of boilerplate function pointers) becomes more approachable. The presence of virtual table support in a wrapper is indeed rare; as one expert noted during review, *"I'm not aware of any [other wrapper] that tackles virtual tables. That would be the standout feature."* Boost.SQLite includes this standout feature, allowing advanced use cases like exposing C++ containers or external data as SQL tables.

- **Hooks (Commit, Rollback, Update, etc.):** Boost.SQLite exposes SQLite's hook interfaces as simple C++ functions. For example, it has `boost::sqlite::update_hook(connection&, Func&&)` which *"installs an update hook"* callback on the connection [34]. Likewise `boost::sqlite::commit_hook(conn, func)` and `rollback_hook` are provided [35] [36]. These take a C++ callable (`func`) which will be invoked at the appropriate time. The library's documentation for `commit_hook` notes: *"The commit hook gets called before a commit is performed. If* `func` *returns true, the commit goes through, otherwise it gets rolled back."* [37]. This matches SQLite's semantics (commit hook can veto a commit by returning non-zero). By wrapping these,

Boost.SQLite allows users to simply pass in a lambda capturing their context, rather than dealing with static function and void pointers as required by `sqlite3_commit_hook`. Similar ease applies to the update hook, which can be set to monitor changes (the `Func` likely receives parameters like table name, rowid, etc., similar to the C API). These hooks are useful for event-driven applications (e.g., invalidating caches or logging changes) and Boost.SQLite making them one-liners to set up is a big convenience. Identifiers here include the free functions: `sqlite::commit_hook`, `sqlite::rollback_hook`, `sqlite::update_hook`, as well as `sqlite::preupdate_hook` (for the newer pre-update hook) [38].

In summary, Boost.SQLite covers the full range of SQLite extensibility: defining custom scalar/aggregate functions in C++, creating virtual table modules, and hooking into database events – all through high-level C++ interfaces. This opens the door for using SQLite in more complex ways (for example, integrating with C++ data structures or reacting to changes) without leaving the comfort of C++ and Boost libraries.

JSON and Variant Integration

Boost.SQLite leverages Boost libraries to provide convenient JSON handling and variant types for SQLite data. As mentioned, SQLite itself treats JSON as text, but Boost.SQLite can integrate with **Boost.JSON** to make JSON usage more natural. The library provides a header `<boost/sqlite/json.hpp>` [39] which, when included, enables JSON support. This likely means you can bind a `boost::json::value` to a query parameter or retrieve a column into a `boost::json::value` directly, with the library doing the serialization/deserialization under the hood. The author notes that *"support for JSON is really simple and completely optional"* [40] – you opt-in by including that header. It's provided mainly *"for convenience"* [41]; it also serves as an example of how one can extend the type system (the documentation suggests it *"shows how easy it is to add your own subtype"* [41]). In Boost.SQLite's design, SQLite's **subtypes** feature is used to tag values with custom types. The JSON integration likely registers a custom subtype for JSON, so that a `boost::json::value` can be stored and retrieved without manual conversion. This allows developers to work with JSON data in SQLite as seamlessly as with regular types, which is especially useful now that JSON is so prevalent in data storage.

On the variant side, Boost.SQLite uses `boost::variant2` (a C++17 variant implementation from Boost) internally for its parameter binding interface [13]. The library's interface for executing statements can accept a variety of C++ types for parameters – integers, doubles, strings, blobs, nulls, etc. Internally, these are all wrapped in a variant type (essentially a `boost::variant2::variant<int64_t, double, std::string, etc.>` corresponding to SQLite's fundamental types). This means when you call `stmt.execute(value1, value2, ...)`, it can handle different types and bind them appropriately. Likewise, query results can be represented as a `boost::sqlite::value` or `boost::sqlite::field` object (both provided by the library) which can convert to C++ types or be accessed as a variant. The library explicitly *"supports variants & json, as those are available through boost"* [4]. This design choice reduces friction when dealing with SQLite's **dynamic typing** – instead of the user having to manually check the column type and convert, the wrapper can supply a variant that the user can `std::visit` or convert with `get<T>()`. It's a modern C++ way to represent the *type union* that is a SQLite value.

In practical terms, a developer using Boost.SQLite can, for example, bind a `boost::json::value` to a `?` parameter in an SQL statement and the library will store it as JSON text (with a subtype flag) in SQLite. When retrieving it, the wrapper might automatically parse it back into a `boost::json::value`. All of this is optional – if you don't include the JSON support, you just handle JSON as text normally. The variant usage is

largely internal, but manifests as flexibility in API: you can supply different C++ types to the same `execute` call or fetch columns without worrying about exact types beforehand. These integrations demonstrate Boost.SQLite's aim to be *"in the idioms of modern C++"*, using C++17 features to enhance the experience of using SQLite.

Packaging and License

Packaging: As of the Boost.SQLite review (2025), the library is not yet part of an official Boost release, so it isn't available via the typical package managers (vcpkg, conan) by default. Users currently build it from source. The project provides a CMake build, producing two libraries: `boost_sqlite` (the main library for core functionality) and `boost_sqlite_ext` (for extension features) [42]. The documentation explains that to use extensions like custom functions and virtual tables, you either link against `boost_sqlite_ext` or include the extension-specific header with a compile-time flag [43]. This split is done to keep the core lightweight for those who don't need the extension points, but it does introduce an extra build step if you do. In other words, basic usage (executing queries, using prepared statements, transactions, etc.) is in the core library, while writing extension modules (custom collations, virtual tables, etc.) is enabled by the separate component. This modular design is sensible, though it means new users must be aware of the two parts. In terms of installation, until it's in a Boost release, one would fetch the repository and add it to their project include path or build it as a dependency. The review notes pointed out that other wrappers are easier to consume (many are header-only or on vcpkg), so packaging is an area Boost.SQLite could improve. However, once accepted into Boost official, it would likely be distributed with Boost and eventually appear in vcpkg as part of Boost, simplifying adoption.

License: Boost.SQLite is released under the **Boost Software License 1.0 (BSL-1.0)**, which is the standard license for Boost libraries. This is a very permissive license, similar to MIT, allowing free use in open-source or proprietary projects. The code files in the repository are all marked with the usual Boost license header (*"Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)"* [44]). This ensures license compatibility with the rest of Boost and is a non-issue for most users (the BSL imposes minimal requirements). In short, there are no unusual licensing restrictions; Boost.SQLite can be used just like any Boost library in both commercial and non-commercial software.

Summary of Boost.SQLite Findings: Boost.SQLite presents a modern C++ interface to SQLite that covers both **basic usage** (executing SQL, prepared statements, transactions) and **advanced SQLite features** (user-defined functions, virtual tables, hooks) within one coherent library. Its standout features include the dual error handling strategy (throwing vs. `error_code` paths) and deep integration points for extending SQLite in C++. The inclusion of JSON and variant support shows an attention to current developer needs. The main tradeoff observed is that, at the time of review, Boost.SQLite is not as plug-and-play as some existing wrappers (which might be header-only or in package managers), but this is expected to improve as it becomes an official Boost library. All functionality is provided under the familiar Boost Software License, ensuring it can be adopted in any project.

Overall, Boost.SQLite aims to "augment" SQLite's C API rather than hide it [14], staying close to SQLite's concepts (one can drop down to raw `sqlite3*` if needed) while offering the convenience and safety of C++ abstractions. It is a promising addition to the Boost family for anyone who works with SQLite in C++, bringing together the low-level power of SQLite with high-level C++ design.

Sources:

- SQLite Documentation – *Prepared Statement Object*: “All SQL must be converted into a prepared statement before it can be run.” (Accessed 2025-09-01) [1]

- SQLite C Intro – *Binding Parameters*: SQLite allows SQL statements to contain parameters which are “bound” to values prior to being evaluated [6]

- SQLite Transactions Documentation: “No reads or writes occur except within a transaction… any command that accesses the database… will automatically start a transaction if one is not already in effect.” (Accessed 2025-09-01) [45]

- SQLiteCpp Library README – Design: “It is designed using the RAII idiom, and throwing exceptions in case of SQLite errors (except in destructors…). Each SQLiteC++ object… is always valid until destroyed.” (Accessed 2025-09-01) [2]

- Boost.SQLite README – Features: *“It includes: typed queries; prepared statements; json support; custom functions (scalar, aggregate, windows); event hooks; virtual tables. sqlite provides an excellent C-API, so this library does not attempt to hide, but to augment it.”* (Accessed 2025-09-01) [46]

- SQLite C Intro – *Extending SQLite*: “The sqlite3_create_module interface is used to register new virtual table implementations… The sqlite3_create_function interface creates new SQL functions – either scalar or aggregate.” [47]

- SQLite C API Reference – *Update Hook*: “The sqlite3_update_hook() interface registers a callback function… to be invoked whenever a row is updated, inserted or deleted in a rowid table.” [10]

- Boost Mailing List (K. Morgenstern) – *Variant usage*: “boost.sqlite uses `boost.variant2` for parameters internally. That’s an implementation detail though.” (Accessed 2025-09-01) [13]

- Boost Mailing List (K. Morgenstern) – *JSON support*: “The support for json is really simple and completely optional… mainly for convenience, however it also shows how easy it is to add your own subtype.” (Accessed 2025-09-01) [40]

- Boost.SQLite Docs – *Comparison/Features*: “boost.sqlite does provide more functionality when it comes to hooks, custom functions & virtual tables… and supports variants & json, as those are available through Boost.” [4]

- Boost.SQLite Reference – *Transaction Guard*: `struct boost::sqlite::transaction` – “A simple transaction guard implementing RAII for transactions.” [23]

- Boost.SQLite Reference – *Savepoint Guard*: `struct boost::sqlite::savepoint` – “A simple transaction guard implementing RAII for savepoints. Savepoints can be used recursively.” [24]

- Boost.SQLite Reference – *create_module*: `boost::sqlite::create_module(conn, name, module, system::error_code &ec, error_info &ei)` – “Register a vtable.” [30]

- Boost.SQLite Reference – *commit_hook*: `bool boost::sqlite::commit_hook(connection &conn, Func &&func)` – *Install a commit hook.* "The commit hook gets called before a commit... If `func` returns true, the commit goes [through], otherwise it gets rolled back." [37]

- Boost.SQLite Reference – *update_hook*: `bool boost::sqlite::update_hook(connection &conn, Func &&func)` – "Install an update hook." [34]

- Boost.SQLite Reference – *Error handling (exceptions)*: *"Exceptions:* `system_error` *from overload without* `ec` *&* `ei` *"* – i.e. a system_error is thrown if you don't use the error_code overload (Accessed 2025-09-01) [15]

- Boost.SQLite Reference – *Error handling (error_code parameters)*: *"ec The system::error_code to capture any possible errors; ei Additional error_info when error occurs."* (Accessed 2025-09-01) [48]

- Boost.SQLite Reference – *Aggregate function requirements*: "`func` *needs to be an object with two functions: void step(State&, span<sqlite::value, N> args); T final(State &). ... When the aggregation is done* `final` *is called and the result is returned to sqlite."* [28]

- Eggs.SQLite (similar wrapper) – *License Declaration*: "Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)." [44]

---

[1] [5] Prepared Statement Object
https://www.sqlite.org/c3ref/stmt.html

[2] GitHub - tiendq/SQLiteCpp: SQLiteCpp is a modern C++ wrapper for SQLite3 library.
https://github.com/tiendq/SQLiteCpp

[3] [11] [13] [39] [40] [41] Boost mailing page: Re: boost.sqlite seeking endorsement
https://listarchives.boost.org/Archives/boost/2024/03/256300.php

[4] [29] boost_sqlite: boost_sqlite
https://klemens.dev/sqlite/

[6] [8] [9] [47] An Introduction To The SQLite C/C++ Interface
https://www.sqlite.org/cintro.html

[7] [45] Transaction
https://www.sqlite.org/lang_transaction.html

[10] Data Change Notification Callbacks
https://sqlite.org/c3ref/update_hook.html

[12] [14] [42] [43] [46] GitHub - klemens-morgenstern/sqlite
https://github.com/klemens-morgenstern/sqlite

[15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [30] [31] [32] [33] [34] [35] [36] [37] [38] [48] boost_sqlite: Reference
https://klemens.dev/sqlite/group__reference.html

44 GitHub - eggs-cpp/eggs-sqlite: Eggs.SQLite is a Modern C++ thin wrapper over SQLite
https://github.com/eggs-cpp/eggs-sqlite