# CS460: Individual project 1 - Relational Operators Deadline: 17.04.2025 23:59

In this project you will implement a query processing engine using Apache Calcite as the foundation. Apache Calcite is an open-source easy-to-extend Apache project, used by many commercial and non-commercial systems. This project covers four topics:

1. the semantics of relational operators
2. the iterator execution model
3. execution using late materialization
4. query optimization rules

We briefly introduce the provided infrastructure AFTER presenting the project's tasks.

## Task 1: Implement a volcano-style tuple-at-a-time engine (35%)

You will implement six basic operators (**Scan**, **Filter**, **Project**, **Join**, **Aggregate** and **Sort**) in a volcano-style tuple-at-a-time operators by extending the provided interface ch.epfl.dias.cs460.rel.early.volcano.Operator, where each operator processes a single tuple-at-a-time. As described in class, each operator in a volcano-style engine requires the implementation of three methods:

- **open()**: Initialize the operator's internal state.
- **next()**: Process and return the next result tuple or NilTuple for $EOF$.
- **close()**: Finalize the execution of the operator and clean up the allocated resources.

You are free to implement the **Join** operator of your preference (we recommend revising the slides from CS-300). The **Scan** operator need to support a row-store storage layout (NSM). Focus on correctness first, and then try to optimize your implementation.

**Assumptions**: In the context of this project, you will store all data in-memory, thus removing the requirement for a buffer manager. In addition, you can assume that the records will consist of objects (Any class in Scala).

**Important!!!** Implement the operators based on the prototypes given in the skeleton code.

**Hint**: We suggest that you start by looking at the documentation and classes that appear in the skeleton code. Rather than try to understand the whole codebase, focus on the task at hand.

# Task 2: Late Materialization (naive) (15%)

In this task, you will implement late materialization using tuple-at-a-time processing with a column-store storage layout (DSM). You do not need to implement a Scan operator for this task, an implementation of LateColumnScan is provided.

Late materialization uses virtual IDs to reconstruct tuples on demand. For example, suppose that our query plan evaluates the relational expression $\sigma_{A.x=5}(A) \bowtie_{A.y=B.y} B$. Without late materialization, the query engine would execute the following steps:

- Scan columns $A.x$ and $A.y$ from table $A$;
- Eliminate rows for which $A.x \neq 5$;
- Join qualifying $(A.x, A.y)$ tuples with table $B$ on $B.y$.

By contrast, with late materialization, the query engine would execute the following steps:

- Scan column $A.x$;
- Eliminate rows for which $A.x \neq 5$;
- For qualifying tuples, fetch $A.y$ to reconstruct $(A.x, A.y)$ tuples;
- Join $(A.x, A.y)$ tuples with table $B$ on $B.y$;

To implement late materialization, we enrich the query engine's tuples so that they contain virtual IDs. We name an enriched tuple LateTuple. It contains a VID and a Tuple of attributes that have already been materialized. A LateTuple is structured as LateTuple(vid : VID, tuple : Tuple).

## Subtask 2.A: Implement Late Materialization primitive operators

In the first subtask you should implement the **Drop** and **Stitch** operators:

**Drop** (ch.epfl.dias.cs460.rel.early.volcano.late.Drop) translates LateTuple to Tuple by dropping the virtual ID. Drop allows interfacing late materialization-based execution with existing non-late materialized operators.

**Stitch** (ch.epfl.dias.cs460.rel.early.volcano.late.Stitch) is a binary operator (has two input operators) and from the stream of LateTuple they provide, it synchronizes the two streams to produce a new stream of LateTuple. More specifically, the two inputs produce LateTuple for the same table but different groups of columns. These streams may or may not miss tuples, due to some pre-filtering. Stitch should find the virtual IDs that exist on both input streams and generate the output as LateTuple that includes the columns of both inputs (first the left input's columns, then the right one's).

**Example**: If the left input produces:
LateTuple(2, Tuple("a")), LateTuple(8, Tuple("c")) ,
and the right input produces:
LateTuple(0, Tuple(3.0)), LateTuple(2, Tuple(6.0)) .
Stitch should produce:
LateTuple(2, Tuple("a", 6.0)) ,
since the only virtual IDs that both input streams share is 2.

### Subtask 2.B: Extend relational operators to support execution on LateTuple data

In the second subtask you should implement a

- **Filter** (ch.epfl.dias.cs460.rel.early.volcano.late.LateFilter)
- **Join** (ch.epfl.dias.cs460.rel.early.volcano.late.LateJoin)
- **Project** (ch.epfl.dias.cs460.rel.early.volcano.late.LateProject)

that directly operate on LateTuple data and. In the case of LateFilter and LateProject, they should preserve virtual IDs.

## Task 3: Query Optimization Rules (50%)

In this task, you will implement new optimization rules to "teach" the query optimizer possible plan transformations. Your rules will be implemented by extending the Apache Calcite optimizer. Then you are going to use these optimization rules to reduce data access in the query plans.

All optimization rules in Calcite inherit from RelOptRule and in (ch.epfl.dias.cs460.rel.early.volcano.late.qo), you can find the skeleton code for the rules you are asked to implement. Specifically, you need to implement onMatchHelper, which computes a new sub-plan that replaces the pattern-matched sub-plan. Note that these rules operate on top of logical operators and not the operators you implemented. The Logical operators are translated into your operators in a subsequent step of planning.

### Subtask 3.A: Implement the Fetch operator

**Fetch** (ch.epfl.dias.cs460.rel.early.volcano.late.Fetch) is a unary operator. It reads a stream of input LateTuple and reconstructs missing columns by directly accessing the corresponding column's values.

For example, assume that we are given column $A.y$: $[5.0, 4.0, 8.0, 1.0]$ .

Also, assume a Fetch operator is used to reconstruct $A.y$ for tuples LateTuple(1, Tuple("a")), LateTuple(2, Tuple("c")) .

Then, the result is LateTuple(1, Tuple("a", 4.0)), LateTuple(2, Tuple("c", 8.0)) .

The advantage of Fetch is that, unlike Stitch, it doesn't have to scan the full column $A.y$.

Also, Fetch optionally receives a list of expressions (evaluator in the handout code) to compute over the reconstructed column. If no expressions are provided, Fetch simply reconstructs the values of the column itself.

**Hint**: to test Fetch, you need to inject it to the plan (Subtask 3.B).

## Subtask 3.B: Implement the Optimization rules

You are asked to implement three optimization rules:

- (ch.epfl.dias.cs460.rel.early.volcano.late.qo.LazyFetchRule) to replace a LateColumnScan → Stitch with a Fetch.
- (ch.epfl.dias.cs460.rel.early.volcano.late.qo.LazyFetchFilterRule) to replace a LateColumnScan → Filter → Stitch with a Filter → Fetch.
- (ch.epfl.dias.cs460.rel.early.volcano.late.qo.LazyFetchProjectRule) to replace a LateColumnScan → Project → Stitch with a Fetch.

An example of LazyFetchRule is shown in Figure 1.

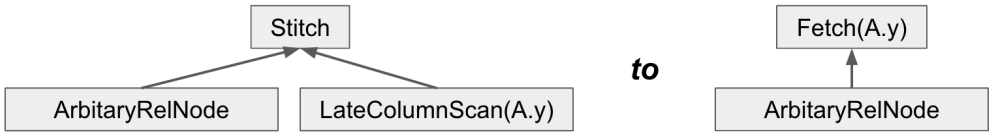


Figure 1: LazyFetchRule transforms the above subplan

# Project setup & grading

Register for access to the repository and grader (DEADLINE 03.03.2025 23:59).

For this project, we are going to be using gitlab.epfl.ch to distribute the skeleton code and accept the project submissions. Thus we require you to register on gitlab.epfl.ch and submit your (sciper → gitlab) username mapping in the following **link**.

After the registration deadline, we will provide you with a personal skeleton codebase `gitlab.epfl.ch/DIAS/COURSES/CS-460/2025/students/Project-1-username`, where username is your GitLab username. In this repository, you are going to be submitting your project and an automatic grader will be picking up your code to provide you feedback on your score.

We suggest you configure your ssh keys and register in gitlab through your user profile settings, SSH Keys (gitlab.epfl.ch/-/user_settings/ssh_keys). The same GitLab page provides further instruction on creating and adding keys.

## Setup your environment

The skeleton codebase is pre-configured for development in (IntelliJ IDEA) and this is the only supported IDE. You are free to use any other IDE and/or IntelliJ version, but it will be your sole responsibility to fix any configuration issues you encounter, including that through other IDEs may not display the provided documentation.

You can get the community edition for free (which is open source) and are eligible for the ultimate version through a free academic license on your EPFL email.

After you install IntelliJ on your machine, from the File menu select *New→Project from Version Control*. Then on the left-hand side panel pick *Repository URL*. On the right-hand side pick:

- Version control: Git
- URL: `gitlab.epfl.ch/DIAS/COURSES/CS-460/2025/students/Project-1-username` or `git@gitlab.epfl.ch:DIAS/COURSES/CS-460/2025/students/Project-1-username`, depending on whether you set up SSH keys (where is your GitLab username)
- Directory: anything you prefer, but in the past we have seen issues with non-ascii code paths (such as french punctuations), spaces and symlinks

IntelliJ will clone your repository and setup the project environment. If you are prompt to import or auto-import the project, please accept. If the JDK is not found, please use IntelliJ's option to *Download JDK*, so that IntelliJ install the JDK in a location that will not change your system settings and the IDE will automatically configure the project paths to point to this JDK.

## Personal repository

Your provided repository is personal, and you are free to push code/branches as you wish. The grader will run on all the branches, but for the final submission, only the main branch will be taken into consideration.

### Additional information and documentation

The skeleton code depends on a library we provide to integrate the project with Apache Calcite, a state-of-the-art query optimizer. Additional information on Calcite can be found on the official site (calcite.apache.org) and in the Calcite documentation site.

Documentation for the helper functions and classes we provide as part of the project-to-Calcite integration code can be found either by browsing the javadoc of the dependency jar (External Libraries/ch.epfl.dias.cs460:base), or by browsing to `http://diascld24.iccluster.epfl.ch:8080/ch/epfl/dias/cs460/helpers.html` WHILE ON VPN.

### Common issues and fixes

**Sources for tests are unresolved in IntelliJ.** Some versions of the IntelliJ Scala plugin have a bug where it cannot locate the sources for test code in our helpers package. e.g., if you command-click to inspect sources for a helper function in the tests, you may see a tasty file instead of the source code. To fix this: in IntelliJ, go to File -> Project Structure -> Libraries. In the list of libraries, select `sbt: ch.epfl.dias.cs460:base_3:2025.1.5:tests:jar` then click the + symbol (add), in file selection window scroll down and click `sbt: ch.epfl.dias.cs460:base_3:2025.1.5:tests-sources:jar`, then click open. Then, in the "Choose Categories of Selected Files" popup, scroll down and select sources, then click OK. Reloading sbt overrides this configuration, so if you reload sbt, you will need to repeat this process.

## Deliverables

We will grade your last commit on your GitLab repository. Your implementation must be on the **main** branch. **You do not submit anything on moodle**. Your repository must contain a **README.md** file.

**Grading:** Keep in mind that we will test your code automatically. Do not change the interfaces of the provided methods.
Any project that fails to conform to the original skeleton code and interfaces will fail in the auto grader, and hence, will be graded as a zero. More specifically, you should not change the function and constructor signatures provided in the skeleton code. You are allowed to add new classes, files and packages, but only under the *scala.app* package. Any code outside the app package will be ignored and not graded. You are free to edit the Main.scala file and/or create new tests, but will be ignored during auto-grading. Tests that timeout will lose all the points for the timed-out test cases, as if they returned wrong results.

Please, do not post any solution or code on Ed or any public forum. The CS-460 team can assist in understanding project/task requirements, and figuring out the correct algorithms but not running/debugging individual code segments.

Please, put anything you want the TAs to know when grading your project in your README.md. If there is nothing special, feel free to leave it blank.