Key Joins

Author: Joel Jakobsson, Vik Fearing, Andreas Karlsson, Arne Roland, Anders Granlund
Status: Discussion
Date: May 28, 2026

Contents

1 Abstract **3**

2 References **3**

3 Discussion **3**

4 Notation **3**

5 Intention **4**

6 Definition **4**
6.1 Why the Definition Captures the Intention . . . 4

7 Specification **5**
7.1 Operand Surfaces and Proof Facts . . . 5
7.2 Syntax . . . 6
7.2.1 Multiple Foreign Keys Between the Same Tables . . . 7
7.2.2 Self-Joins . . . 8
7.2.3 Join-Local FILTER . . . 8
7.3 Compile-Time Verification . . . 10
7.3.1 Verifying Condition 1 (uniqueness) . . . 10
7.3.2 Verifying Condition 2 (containment) . . . 11
7.3.3 Verifying Condition 3 (null handling) . . . 13
7.3.4 When the DBMS Cannot Form a Proof . . . 14
7.4 Propagation Through Derived Tables . . . 16
7.4.1 Validity in Derived Tables . . . 16
7.4.2 BUNC-Sets . . . 17
7.4.3 How Properties Compose Through Layers . . . 17
7.4.4 A View That Works . . . 18
7.4.5 A CTE That Works . . . 18
7.4.6 LATERAL Derived Tables . . . 18
7.4.7 Breaking Uniqueness . . . 19
7.4.8 Restoring Uniqueness with GROUP BY . . . 19
7.4.9 Null-extension . . . 20
7.4.10 Nullable UNIQUE Constraints . . . 21
7.4.11 Inner Key Joins with Known Not Nullable Referencing Columns . . . 23
7.4.12 Set Operations . . . 23
7.4.13 Filtered Views and the PK Side . . . 23
7.4.14 Safe View Composition . . . 24
7.5 Constraint Requirements . . . 25
7.5.1 Exclusion of Temporal Referential Constraints . . . 26
7.6 Schema Changes and Dependent Objects . . . 26
7.6.1 Real-World Consequences . . . 26

7.6.2 Schema Migration Breaking Uniqueness . . . 27
7.6.3 Data Maintenance Breaking Row-Preservation . . . 28
7.6.4 Error Messages and Privilege Scoping . . . 29
7.7 Soundness of the Specification . . . 29

8 Design Rationale **30**
8.1 Compile-Time Verification . . . 30
8.1.1 The Query Author's Contract . . . 30
8.1.2 Runtime Checks Are Insufficient . . . 30
8.1.3 Key Joins Aid Testing . . . 30
8.1.4 Compile-Time Verification Enables Composition . . . 31
8.2 Why Arrow Syntax . . . 31
8.3 Why Arrow Direction Is Fixed by the FK, Not Chosen by the Author . . . 31
8.4 Why Column Lists Instead of Constraint Names . . . 31
8.5 Why the Joined Table's Column List Takes No Alias . . . 32
8.6 Why Explicit Columns Instead of Inference . . . 32
8.7 Indirect Foreign Keys Are Not Supported . . . 32
8.8 Benefits . . . 33
8.9 Readability at Scale . . . 33

9 Conclusion **35**

1. Abstract

A change proposal for key joins: a syntax for equijoins that follow referential constraints. A key join is written with an arrow indicating the direction of the foreign key relationship. The Database Management System (DBMS) verifies at compile time that a matching referential constraint exists and that the join is semantically valid. Key joins work through views, CTEs, and other derived tables, provided that the underlying referential relationship is preserved through the derivation.

2. References

[Foundation] ISO/IEC 9075-2, SQL/Foundation (IWD)

[Schemata] ISO/IEC 9075-11, SQL/Schemata (IWD)

3. Discussion

SQL equijoins are expressed as value comparisons: `A.x = B.y`. This says that two columns share a comparable value, but nothing more. It does not say whether one column references the other, which direction that reference goes, whether the referenced side is unique, or whether the referencing side's values are contained within the referenced side's values.

In practice, many equijoins in application code follow referential constraints. The query author intends to navigate from one table to another along a defined referential path, but the query itself does not express this. The intent lives in naming conventions, comments, or the query author's mental model. Nothing in the query would break if the schema changed in a way that invalidated the assumed relationship.

Key joins make this intent explicit. The query names the columns on both sides and indicates the direction with an arrow. The DBMS verifies at compile time that the join faithfully follows the declared relationship. If the schema changes in a way that invalidates the join, the query fails with a clear error rather than silently producing wrong results.

The example error messages shown below use wording from the PostgreSQL prototype implementation and are illustrative, not normative.

4. Notation

A **key join** is an equijoin between two tables, the **referencing table** and the **referenced table**, where the join predicate compares **referencing columns** in the referencing table with **referenced columns** in the referenced table. The two column lists have the same arity and pairwise comparable types.

Like an equijoin, a key join has two input operands, a left and a right; one is the referencing table and the other the referenced table. When key joins are chained in a query, the right operand of each is the *newly introduced table* for that key join.

Throughout this paper, "FK" abbreviates "referencing" and "PK" abbreviates "referenced" (e.g. "FK side", "PK columns"). The "PK" abbreviation does not imply a primary key; the referenced side may be backed by either a PRIMARY KEY or a UNIQUE constraint.

5. Intention

The intention of the key join syntax is to make a referential equijoin explicit, verifiable, and locally readable. A key join enriches a referencing table with a referenced table by following a declared referential constraint. The referencing rows are preserved: each appears in the result exactly once. Each all-non-null referencing key is enriched with its unique referenced row.

The intent is asymmetric. The referencing table is the table whose rows are guaranteed to survive the join; the referenced table is consulted for a unique matching row but is not itself preserved by the key-join guarantee. The syntax lets the reader identify the referencing and referenced sides, and the compared columns, directly from the query.

The DBMS verifies the required facts at compile time from declarations and query structure. The proof rules are sound rather than complete: a mathematically valid key join may still be rejected when the specified rules cannot derive the necessary facts.

6. Definition

The conditions under which an equijoin is a key join are stated in terms of two multisets:

- The **referenced multiset** is the projection of the referenced table onto the referenced columns, restricted to rows where every referenced column is non-null.
- The **referencing multiset** is the projection of the referencing table onto the referencing columns.

Such an equijoin is a **key join** if and only if all of the following conditions hold:

1. Every element of the referenced multiset has a multiplicity of exactly 1 (one).
2. For every element of the referencing multiset where no value is null, the referenced multiset contains that element.
3. Either
   (a) no element of the referencing multiset contains a null value, or
   (b) the referencing table is preserved via an outer join type.

6.1. Why the Definition Captures the Intention

The Intention requires that every row of the referencing table appears in the result exactly once, and that every all-non-null referencing key is enriched with exactly one referenced row.

- **Condition 1 prevents duplication:** If every referenced multiset element has multiplicity one, no referencing row finds more than one match in the equijoin, so no referencing row is duplicated. The same uniqueness supplies the singularity of the referenced row used for enrichment.
- **Conditions 2 and 3 together prevent loss and supply matches:** Condition 2 ensures every all-non-null referencing key has a match in the referenced multiset, so no referencing row with an all-non-null key is dropped. Together with Condition 1, that match is unique. Duplicate referencing keys are allowed: several referencing rows may match the same referenced value, provided the referenced value itself is unique. Condition 3 handles the null case: either the referencing multiset has no null elements (3a), or the join type preserves the referencing rows whose keys are null (3b). Without one of these disjuncts, an inner equijoin on a nullable referencing key would silently drop the null-keyed referencing rows.

The conditions are sufficient for preservation and enrichment. They are also individually necessary: any one of them failing exhibits a referencing row that is lost, duplicated, or not enriched with a unique referenced row. We do not claim the conditions are the unique characterization of preservation and enrichment, only that each is sufficient and individually necessary.

7. Specification

The Specification realises the Definition as an SQL operator with compile-time verification. It comprises:

- **Syntax** (Section 7.2): a syntax for declaring a key join, naming the referencing and referenced sides and their column lists.
- **Compile-time verification** (Section 7.3): verification rules that accept a key join only when the DBMS can prove the three conditions of the Definition against the join's intermediate result. One sub-subsection addresses each condition; a fourth covers cases where no proof can be formed.
- **Propagation through derived tables** (Section 7.4): rules that carry the proof obligations through views, common table expressions (CTEs), and subqueries.
- **Constraint requirements** (Section 7.5): the schema-level declarations (PRIMARY KEY, UNIQUE, FOREIGN KEY, NOT NULL) the Specification relies on.
- **Schema changes** (Section 7.6): how dependent objects, especially views, interact with constraint changes that would invalidate a key join.
- **Soundness** (Section 7.7): why every key join accepted by the verification rules satisfies the Definition.

7.1. Operand Surfaces and Proof Facts

A "surface" is any FROM-clause item – base table, subquery, CTE, join tree, function, view, TABLESAMPLE, etc. Each maps to a `<table reference>`. Currently only base tables, subqueries, CTEs, and join trees produce facts. A surface carries four kinds of facts, all schema-derived.

The two surfaces supplied to a key join operator are its "operand surfaces".

Unique fact. A set of columns known to be unique. No two rows share the same tuple of values in those columns (nulls don't violate uniqueness).

Not null fact. A single column known to reject nulls.

Containment and coverage fact. Both are based on the same predicate between two surfaces S and S' on column lists C and C':

> For every row R in S, either $P_C(R)$ contains a null, or there exists a row R' in S' such that $P_C(R) \subseteq_{\text{pointwise}} P_{C'}(R')$.

A containment fact lives on S (the referencing side). A coverage fact is the same predicate, indexed on S' (the referenced side).

A catalog FK from $A(C)$ to $B(C')$ yields a containment fact on A. A unique index on B yields a coverage fact on B. The FK also yields a coverage fact on A.

The two differ in how they handle filters:

- Filtering S preserves containment (fewer rows, still a subset), so the fact stays active. The filter is optionally recorded as `filterConjunct`.
- Filtering S' can break coverage – removed rows might have been the only match. Filters MUST be recorded as `filterConjuncts`; unrecognized filters deactivate the fact.

When both sides are filtered, the proof remaps the coverage-side filter through the FK column pairing and checks that the containment side recorded a matching filter. This is matched filter evidence – part of proving condition 2 when both sides carry filters.

7.2. Syntax

Consider a schema for an organization:

```
CREATE TABLE departments (
    dept_id   INTEGER PRIMARY KEY,
    name      CHARACTER VARYING(100),
    active    BOOLEAN DEFAULT TRUE NOT NULL
);

CREATE TABLE employees (
    emp_id      INTEGER PRIMARY KEY,
    name        CHARACTER VARYING(100),
    dept_id     INTEGER NOT NULL REFERENCES departments (dept_id),
    home_dept   INTEGER REFERENCES departments (dept_id),
    manager_id  INTEGER REFERENCES employees (emp_id)
);
```

A query that retrieves an employee's department using a key join:

```
SELECT e.name, d.name AS department
FROM employees AS e
JOIN departments AS d FOR KEY (dept_id) <- e (dept_id)
WHERE e.emp_id = 4017;
```

The arrow `<-` points from the FK table (`employees`) to the PK table (`departments`). This declares: "join `departments` to `employees` along the referential constraint from `employees.dept_id` to `departments.dept_id`."

The arrow can point either way. A query starting from departments:

```
SELECT d.name AS department, e.name
FROM departments AS d
JOIN employees AS e FOR KEY (dept_id) -> d (dept_id)
WHERE d.dept_id = 12;
```

Here `->` points from the FK table (`employees`, being joined) to the PK table (`departments`, already in the FROM clause). This returns all employees in department 12. The direction always reflects the referential constraint, regardless of which table appears first in the query.

A key join is written as a `<join specification>`, alongside ON and USING:

```
<table reference> [ <join type> ]
    JOIN right_rel FOR KEY (right_col [, ...]) -> left_rel (left_col [, ...])

<table reference> [ <join type> ]
    JOIN right_rel FOR KEY (right_col [, ...]) <- left_rel (left_col [, ...])
```

The column list immediately after `FOR KEY` names columns exposed by the syntactic right operand of the join. The relation name after the arrow identifies a relation visible from the left operand. The `->` form is used when the right operand is the FK side; the `<-` form is used when the right operand is the PK side. Because a key join is a `<join specification>`, it can appear anywhere a `<joined table>` can appear, including `<query expression>`s used within `<insert statement>`s, `<merge statement>`s, `<declare cursor>`, and subqueries.

The columns in each list are matched positionally: the first column on the left corresponds to the first column on the right, and so on. The order need not match the order in the constraint definition; only the pairwise correspondence between the two lists matters.

A key join is a `<join specification>`, so it does not apply to `<cross join>`, which has no `<join specification>`.

A key join determines the join condition but does not merge output columns. Its output columns are the same as for `JOIN ... ON`, not the reduced column set produced by `JOIN ... USING`. A `USING (...)` form with a join-local `FILTER` could be defined mechanically, but this proposal keeps the column mapping explicit by naming both the referenced and referencing columns.

Key joins may be combined with LEFT, RIGHT, and FULL join types. The key join specifies the join condition (which referential constraint is being followed and which columns correspond) while the join type controls the result shape. These are orthogonal, just as ON and join type are orthogonal in existing SQL.

When used with an outer join, the null-extension rules apply as usual. A LEFT key join preserves all rows from the left-hand table, null-extending columns from the right-hand table when no match is found. The DBMS still verifies the referential constraint at compile time; what changes is only the treatment of unmatched rows in the result.

The most common use of an outer key join involves nullable FK columns. A referential constraint permits NULL values in the FK columns: a NULL FK value does not violate the constraint, but it does not match any PK row. An inner key join excludes these rows; a left key join preserves them:

```
-- Employees without a home department appear with NULL home_department name:
SELECT e.name, d.name AS home_department
FROM employees AS e
LEFT JOIN departments AS d FOR KEY (dept_id) <- e (home_dept);
```

7.2.1. Multiple Foreign Keys Between the Same Tables

When two tables are related by more than one referential constraint, the column lists disambiguate which relationship the join follows:

```
-- Join on the work department:
SELECT e.name, d.name AS work_department
FROM employees AS e
JOIN departments AS d FOR KEY (dept_id) <- e (dept_id);

-- Join on the home department:
SELECT e.name, d.name AS home_department
FROM employees AS e
LEFT JOIN departments AS d FOR KEY (dept_id) <- e (home_dept);
```

The column lists make it unambiguous which referential constraint is being followed.

7.2.2. Self-Joins

A table may reference itself. In the schema above, `employees.manager_id` references `employees.emp_id`. A key join can follow this self-referencing constraint by aliasing the table twice:

```
SELECT e.name, m.name AS manager
FROM employees AS e
LEFT JOIN employees AS m FOR KEY (emp_id) <- e (manager_id)
WHERE e.emp_id = 4017;
```

The arrow points from `e` (the FK side, via `manager_id`) to `m` (the PK side, via `emp_id`). The two aliases form separate operand surfaces, even though they resolve to the same base table.

7.2.3. Join-Local FILTER

SQL developers commonly add predicates to the ON clause of a join beyond the equijoin condition:

```
SELECT e.name, d.name AS department
FROM employees AS e
LEFT JOIN departments AS d ON d.dept_id = e.dept_id
                          AND d.active;
```

This filters which rows qualify as a match *during* the join. For a LEFT JOIN, non-matching rows are null-extended rather than eliminated, which is semantically different from filtering in WHERE.

`USING` and `FOR KEY` both replace the ON clause with a more specialized join specification. That leaves no ON-clause position for additional match-time predicates. For inner joins, this is often not a problem: the predicate can move to WHERE with identical semantics:

```
-- Inner join: WHERE is equivalent to an ON-clause filter.
SELECT e.name, d.name AS department
FROM employees AS e
JOIN departments AS d FOR KEY (dept_id) <- e (dept_id)
WHERE d.active;
```

For outer joins, WHERE changes the meaning. Moving the predicate out of ON converts a LEFT JOIN into an effective inner join:

```
-- Not equivalent: eliminates employees in inactive departments.
SELECT e.name, d.name AS department
FROM employees AS e
LEFT JOIN departments AS d FOR KEY (dept_id) <- e (dept_id)
WHERE d.active;
```

A derived table over active departments does not help for this example. The filter removes referenced rows on a column outside the key mapping, so the key join is rejected:

```
-- Rejected: filtered derived table lacks row coverage.
SELECT e.name, d.name AS department
FROM employees AS e
LEFT JOIN (SELECT * FROM departments WHERE active) AS d
    FOR KEY (dept_id) <- e (dept_id);

-- ERROR:  key join from referencing relation e to referenced relation d cannot be proven
-- LINE 5:     FOR KEY (dept_id) <- e (dept_id);
--             ^
-- DETAIL:  Not every e (dept_id) value can be proven to have a matching d row. Referenced
    relation d is filtered before this key join.
```

To address this, `FOR KEY` joins accept an optional join-local `<filter clause>`, using the existing production defined for aggregate functions (Subclause 10.9, "`<aggregate function>`"):

```
<filter clause> ::= FILTER <left paren> WHERE <search condition> <right paren>
```

When appended to a key join, the filter is ANDed with the equijoin condition derived from the referential constraint. The key join is still proven before the filter is applied; the filter narrows which rows qualify as a match at runtime:

```
SELECT e.name, d.name AS department
FROM employees AS e
LEFT JOIN departments AS d FOR KEY (dept_id) <- e (dept_id)
                           FILTER (WHERE d.active);
```

This preserves all employees. Departments that exist but are not active are null-extended, exactly as with the traditional ON-clause form. The compile-time verification is unchanged; the filter adds a data condition without weakening it.

The join-local filter is not proof evidence for the key join. It cannot make a nullable FK column known not null, and it cannot repair a PK-side derived table that removed rows before the join. Those facts must be proven from the two operand surfaces before the join-local filter is considered.

The filter clause is permitted with all join types. For inner joins it is a convenience equivalent to WHERE; for outer joins it is the way to express match-time filtering within a specialized join specification. Each key join in a query can carry its own filter clause:

```
SELECT e.name,
       d.name  AS department,
       m.name  AS manager,
       md.name AS manager_department
FROM employees AS e
JOIN departments AS d FOR KEY (dept_id) <- e (dept_id)
                      FILTER (WHERE d.active)
LEFT JOIN employees AS m FOR KEY (emp_id) <- e (manager_id)
LEFT JOIN departments AS md FOR KEY (dept_id) <- m (dept_id)
                            FILTER (WHERE md.active);
```

Each FILTER applies only to its own join. An employee whose department is inactive gets a null-extended department, but the manager join is unchanged. If the manager's department is inactive, only `manager_department` is null-extended. The output facts of that join are then propagated normally, so a later key join can use or reject them in the same way as facts produced by any other join.

7.3. Compile-Time Verification

The Specification is sound: the DBMS accepts a key join only when all three conditions of the Definition can be proven against the intermediate result at the point the join is evaluated. The DBMS rejects whenever it cannot discharge the proof obligation for at least one condition, whether by direct refutation, absent declaration, or opacity of a structural construct.

The next three subsections verify each condition in turn; the fourth covers cases where no proof can be formed.

7.3.1. Verifying Condition 1 (uniqueness)

Condition 1 requires the referenced multiset to have no duplicates. The check is against the intermediate result at the join point, not against the base table alone: a prior join may have duplicated the referenced rows, so a column that is unique in its base table need not remain unique downstream.

Suppose a query author wants per-order totals across the order's items and payments, combined in a single query. A direct ON-joins formulation is a fan trap, joining the referenced `orders` table to two of its referencing tables and silently multiplying rows:

```
-- Wrong at runtime: doubles each total (fan trap).
SELECT o.id,
       SUM(oi.amount) AS item_total,
       SUM(p.amount)  AS payment_total
FROM orders AS o
LEFT JOIN order_items AS oi ON oi.order_id = o.id
LEFT JOIN payments    AS p  ON p.order_id  = o.id
GROUP BY o.id;
```

The query runs without complaint and produces a Cartesian product between `order_items` and `payments` within each order. An order with two items and two payments appears four times, and both aggregates are doubled. GROUP BY at the outer level cannot un-double the SUMs: the duplication has already inflated each group's contribution before the grouping happens.

This same aggregate fan-trap pitfall appears in a PostgreSQL mailing-list thread.[1]

Written with key joins, the same query is rejected at compile time:

[1] PostgreSQL pgsql-general thread "Avoiding double-counting in aggregates with more than one join?", 2016-11-18, https://www.postgresql.org/message-id/flat/86b9ec78-925c-1935-bc9d-6bad4ceb1f40@illuminatedcomputing.com.

```
-- Rejected at compile time.
SELECT o.id,
       SUM(oi.amount) AS item_total,
       SUM(p.amount)  AS payment_total
FROM orders AS o
LEFT JOIN order_items AS oi FOR KEY (order_id) -> o (id)
LEFT JOIN payments    AS p  FOR KEY (order_id) -> o (id)
GROUP BY o.id;

-- ERROR:  key join from referencing relation p to referenced relation o cannot be proven
-- LINE 7: LEFT JOIN payments    AS p  FOR KEY (order_id) -> o (id)
--                                     ^
-- DETAIL:  Referenced columns o (id) are not proven unique. A preceding join may duplicate
     rows from referenced relation o.
```

Both key joins reach `orders` on the PK side. After the first key join, each `orders` row has been duplicated once per matching `order_items` row, so `o.id` is no longer unique in the intermediate result. The second key join is rejected against that intermediate result, not against the base table's own primary key.

The repair is to pre-aggregate each FK-side table first, then join the pre-aggregated results to `orders`:

```
SELECT o.id,
       oi.item_total,
       p.payment_total
FROM orders AS o
LEFT JOIN (
    SELECT order_id, SUM(amount) AS item_total
    FROM order_items
    GROUP BY order_id
) AS oi FOR KEY (order_id) -> o (id)
LEFT JOIN (
    SELECT order_id, SUM(amount) AS payment_total
    FROM payments
    GROUP BY order_id
) AS p FOR KEY (order_id) -> o (id);
```

Each subquery produces at most one row per `order_id`, so neither LEFT JOIN can duplicate the `orders` row. Each key join sees a PK side whose `id` remains unique, both joins are accepted, and the result has one row per order with expected, non-duplicated totals.

7.3.2. Verifying Condition 2 (containment)

Condition 2 requires each distinct all-non-null element of the referencing multiset to appear in the referenced multiset. The DBMS verifies this in two parts: the named columns must match a declared referential constraint between the two base tables; and any derivation on the PK side must provide row coverage for the PK values that the FK side may need. The second part is satisfied when the PK side applies no row-removing filter to the relevant key values, or when every direct key-equality filter on the PK side has a matching direct key-equality filter on the corresponding FK columns. Extra filters on the FK side are allowed; they only narrow the rows that need to be matched.

A schema designer working on a multi-tenant application declares two views, intending each query to see only the current tenant's rows. The schema uses composite primary keys with a leading `tenant_id` column matched against `CURRENT_USER`:

```
CREATE TABLE customers (
    tenant_id  CHARACTER VARYING(128),
    id         INTEGER,
    name       CHARACTER VARYING(100),
    PRIMARY KEY (tenant_id, id)
);

CREATE TABLE orders (
    tenant_id    CHARACTER VARYING(128),
    id           INTEGER,
    customer_id  INTEGER NOT NULL,
    amount       NUMERIC(10,2),
    PRIMARY KEY (tenant_id, id),
    FOREIGN KEY (tenant_id, customer_id) REFERENCES customers (tenant_id, id)
);

CREATE VIEW tenant_customers AS
    SELECT * FROM customers WHERE tenant_id = CURRENT_USER;

-- The schema designer forgot the filter:
CREATE VIEW tenant_orders AS
    SELECT * FROM orders;
```

A query author writes the key join through the two views:

```
-- Rejected at compile time.
SELECT o.id, c.name
FROM tenant_orders     AS o
JOIN tenant_customers  AS c
    FOR KEY (tenant_id, id) <- o (tenant_id, customer_id);

-- ERROR:  key join from referencing relation o to referenced relation c cannot be proven
-- LINE 5:     FOR KEY (tenant_id, id) <- o (tenant_id, customer_id);
--             ^
-- DETAIL:  Not every o (tenant_id, customer_id) value can be proven to have a matching c
    row. Referenced relation c has a filter that is not matched by referencing relation o.
```

`tenant_customers` restricts to the current tenant's customers; `tenant_orders` contains every tenant's orders. The non-null `(tenant_id, customer_id)` values in `tenant_orders` do not all appear among the `(tenant_id, id)` values in `tenant_customers`: an order whose `tenant_id` is not the current tenant's would have no match in `tenant_customers`, and an inner equijoin would silently drop it. The DBMS therefore cannot prove Condition 2.

The repair is to give `tenant_orders` the same filter:

```
DROP VIEW tenant_orders;
CREATE VIEW tenant_orders AS
    SELECT * FROM orders WHERE tenant_id = CURRENT_USER;
```

Both views now apply matching direct equality predicates on the `tenant_id` prefix of the foreign key. Each view is restricted to the current tenant's rows, and within that subset every order's `(tenant_id, customer_id)` has a matching `tenant_customers` value `(tenant_id, id)`. The key join is accepted:

```
-- Accepted at compile time.
SELECT o.id, c.name
FROM tenant_orders     AS o
JOIN tenant_customers  AS c
    FOR KEY (tenant_id, id) <- o (tenant_id, customer_id);
```

The example shows Condition 2 from two sides. The first configuration leaves containment unproven: the PK side is filtered on its key columns without matching direct key-equality evidence on the FK side, so containment is not guaranteed. The second supplies the missing evidence: when every row-removing key filter on the PK side is matched on the FK columns, containment holds within the filtered subset and the key join is accepted. A join-local `FILTER (WHERE ...)` clause would not repair the first query, because that filter is evaluated as part of the join and is not operand-surface proof evidence. A schema designer migrating an application to key joins surfaces inconsistencies of this kind at compile time, where an `ON`-join would have left them silent.

7.3.3. Verifying Condition 3 (null handling)

Condition 3 requires that either the referencing multiset contains no null values (3a), or the join type preserves the referencing table (3b). The DBMS discharges 3a from NOT NULL constraints on the referencing columns, accounting for null-extension introduced by upstream outer joins in the intermediate result. It discharges 3b by inspecting the join type. If neither holds, the key join is rejected.

For the examples below, assume `orders` has a nullable `customer_id` that references `customers (id)`, and `customers` has a NOT NULL `customer_type_id` that references `customer_types (id)`.

An inner key join from `orders` to `customers` fails Condition 3:

```
SELECT *
FROM orders AS o
JOIN customers AS c FOR KEY (id) <- o (customer_id)
WHERE o.id = 100;

-- ERROR:  key join from referencing relation o to referenced relation c cannot be proven
-- LINE 3: JOIN customers AS c FOR KEY (id) <- o (customer_id)
--                             ^
-- DETAIL:  This inner join could filter rows from o. Referencing columns o (customer_id)
    can be null.
```

3a fails because `customer_id` is nullable; 3b fails because the join type is inner. An order with NULL `customer_id` would be silently dropped by the equijoin. The repair is to use an outer join that preserves `orders`:

```
SELECT *
FROM orders AS o
LEFT JOIN customers AS c FOR KEY (id) <- o (customer_id)
WHERE o.id = 100;
```

The LEFT JOIN preserves `orders`, satisfying 3b. Orders without a customer are returned with the customer columns null-extended.

The check applies at every key join in a chain, against the intermediate result at that point. A query that joins orders, customers, and customer_types in a chain can fail Condition 3 at the second join even when every column is properly declared:

```
SELECT *
FROM orders AS o
LEFT JOIN customers AS c  FOR KEY (id) <- o (customer_id)
JOIN customer_types AS ct FOR KEY (id) <- c (customer_type_id)
WHERE o.id = 100;

-- ERROR:  key join from referencing relation c to referenced relation ct cannot be proven
-- LINE 4: JOIN customer_types AS ct FOR KEY (id) <- c (customer_type_id)
--                                   ^
-- DETAIL:  This inner join could filter rows from c. Referencing columns c (
    customer_type_id) can be null because a preceding outer join can null-extend the
    referencing side.
```

This same nullable-side inner-join pitfall appears in a PostgreSQL mailing-list thread.[2]

Although `customers.customer_type_id` is declared NOT NULL in the base table, the LEFT JOIN to `customers` null-extends it for orders without a matching customer. The second key join sees a referencing column that is no longer known not null in the intermediate result. 3a fails again because of the upstream null-extension; 3b fails because the second join is inner. The simplest repair is to make the second join outer too:

```
SELECT *
FROM orders AS o
LEFT JOIN customers      AS c  FOR KEY (id) <- o (customer_id)
LEFT JOIN customer_types AS ct FOR KEY (id) <- c (customer_type_id)
WHERE o.id = 100;
```

An alternative is to group `customers` and `customer_types` into a parenthesized join first, so that the LEFT JOIN to `orders` null-extends a fully resolved intermediate; the inner key join can then remain inner. Both repairs satisfy Condition 3 at every join.

The propagation of not-nullness through chains, including its loss when null-extension can introduce NULLs, is treated more fully in Section 7.4.

7.3.4. When the DBMS Cannot Form a Proof

Some queries fail verification not because they violate a condition outright, but because the DBMS cannot form a proof at all. A key join is accepted only when all three conditions can be proven, and these queries leave at least one condition unproven. Three causes arise: the named columns may not match any declared referential constraint, the columns may not trace to a single base table, or a structural construct may make the proof obligation untrackable. Column traceability and structural opacity are handled by the propagation rules of Section 7.4. The example below shows the first cause: a referential constraint exists between the base tables, but the named columns do not match it.

Consider a hotel booking system:

[2] PostgreSQL pgsql-general thread “LEFT and RIGHT JOIN”, 2012-06-29, https://www.postgresql.org/message-id/flat/CAH3i69msZTFDcnmgW0vJm6YjVMni+rW=xBgWtJNoKGee26JpFg@mail.gmail.com.

```
CREATE TABLE hotels (
    hotel_id    INTEGER PRIMARY KEY,
    name        CHARACTER VARYING(100)
);

CREATE TABLE rooms (
    hotel_id    INTEGER REFERENCES hotels (hotel_id),
    room_number INTEGER NOT NULL,
    room_type   CHARACTER VARYING(50),
    rate        DECIMAL(10,2),
    PRIMARY KEY (hotel_id, room_number)
);

CREATE TABLE reservations (
    reservation_id  INTEGER PRIMARY KEY,
    hotel_id        INTEGER NOT NULL,
    room_number     INTEGER NOT NULL,
    guest_name      CHARACTER VARYING(100),
    check_in        DATE,
    check_out       DATE,
    FOREIGN KEY (hotel_id, room_number)
        REFERENCES rooms (hotel_id, room_number)
);
```

The referential constraint from `reservations` to `rooms` is composite: `(hotel_id, room_number)`. A developer who overlooks this writes:

```
SELECT r.room_type, r.rate, res.guest_name
FROM reservations AS res
JOIN rooms AS r ON r.room_number = res.room_number
WHERE res.reservation_id = 16354;
```

During development and testing, there is only one hotel. The query returns one row and appears correct. In production, multiple hotels share room numbers, and the query silently returns duplicates:

```
 room_type | rate   | guest_name
-----------+--------+-----------
 Single    | 100.00 | John Doe
 Suite     | 250.00 | John Doe
(2 rows)
```

Rewriting the join as a key join exposes the bug at compile time:

```
SELECT r.room_type, r.rate, res.guest_name
FROM reservations AS res
JOIN rooms AS r FOR KEY (room_number) <- res (room_number);

-- ERROR:  key join from referencing relation res to referenced relation r cannot be proven
-- LINE 3: JOIN rooms AS r FOR KEY (room_number) <- res (room_number);
--                         ^
-- DETAIL:  There is no matching foreign key constraint for res (room_number) referencing r
     (room_number).
```

The named columns (`room_number`) alone do not match the declared referential constraint between `reservations` and `rooms`. The DBMS has no declaration to start validation, and rejects the key join. The error leads the developer to discover the composite key and write:

```
SELECT r.room_type, r.rate, res.guest_name
FROM reservations AS res
JOIN rooms AS r FOR KEY (hotel_id, room_number) <- res (hotel_id, room_number)
WHERE res.reservation_id = 16354;
```

7.4. Propagation Through Derived Tables

A query author should not need to know whether they are joining a base table or a view. If a schema designer creates a view over a base table that participates in a referential constraint, the view should be usable in a key join, provided it preserves the properties that make the join valid.

This reflects a separation of duties. The schema designer is responsible for constructing views that preserve the required referential properties. The query author uses them without needing to inspect how they are built.

7.4.1. Validity in Derived Tables

For a key join through a derived table, the columns named on each side must trace to base table columns, and a matching referential constraint must exist between those base tables. On the PK side, the derived table must expose a unique PK value for each all-non-null FK value that can reach the key join.

A key join is an equijoin and NULL never equals anything. The PK-side derived table therefore does not have to reproduce base table row multiplicities. Instead, it supplies Condition 2's value-containment evidence. Uniqueness separately ensures that each matching PK value appears at most once.

This requires that each column named in the key join traces to exactly one base table column ("column traceability" via Subclause 7.16, "`<query specification>`"). Expressions, CASE, COALESCE, and casts break traceability. All columns on each side must trace to the same base table instance.

Given traceability, the PK-side proof decomposes into two properties that the DBMS can verify at compile time and propagate through derived tables:

1. **Row coverage** (nothing needed is missing). The PK side must expose every PK value that an all-non-null FK value can require. This is a value-containment proof, not a statement about base table multiplicity.

   A WHERE clause, HAVING clause, TABLESAMPLE, OFFSET, FETCH FIRST, or set operation can remove rows and break row coverage. Recursive CTEs, LATERAL derived tables, set functions in the `<select list>`, subtable/supertable references, and joins that do not use FOR KEY are conservative proof barriers. A FOR KEY inner join with known not null FK columns

preserves row coverage for the FK side: the constraint guarantees a match for every row, so no rows are lost from that side.

Row coverage can also be conditional on matched filters. A PK-side derived table that applies a WHERE filter on direct key columns exposes conditional row coverage for that filter, provided the FK-side derived table applies every corresponding direct key-equality filter on the FK columns. Additional conjuncts on the FK side are allowed; they only narrow the FK side further. A join-local `FILTER (WHERE ...)` clause is not such evidence, because it applies to the join result rather than to the operand surface. This is illustrated in Section 7.3.2.

Projection and FOR KEY outer joins that preserve the relevant side preserve row coverage. GROUP BY and DISTINCT preserve an existing row-coverage fact only when the row-collapse key covers the same key positions under the same equality identity. Extra grouping or distinct columns do not invalidate the proof. ROLLUP, CUBE, GROUPING SETS, set operations, OFFSET, and FETCH FIRST are not used as row-coverage proof.

2. **Uniqueness** (nothing duplicated). The PK columns must still be unique in the derived table. This is tracked with BUNC-sets, described below. Joins can break uniqueness by duplicating rows, but GROUP BY on the PK columns can restore it. A key join can also preserve a referenced-side uniqueness fact when the referencing side is proven unique on the corresponding FK columns; otherwise possible fanout inactivates that uniqueness fact. ROLLUP, CUBE, and GROUPING SETS are treated conservatively because the same values can appear at multiple aggregation levels.

Null-extension on the PK side is inert. An outer join can introduce rows where the PK columns are NULL, but NULL never equals anything in an equijoin, so those rows cannot match an FK row and cannot break either property above. Instead of exposing a positive null-extension fact, the DBMS tracks not-nullness compositionally through derived tables; a null-extending join removes not-null evidence for the affected columns. That evidence is consulted on the FK side, not the PK side.

On the FK side, column traceability is required. The detailed treatment of nullable FK columns appeared in Section 7.3 under Condition 3; the same rules apply when the FK side is a derived table. Filtering, grouping, and deduplication on the FK side do not invalidate the referential relationship; they merely reduce the set of rows to be matched.

7.4.2. BUNC-Sets

A key join needs the referenced columns to identify at most one matching referenced row for each all-non-null referencing key. It does not need the referenced columns themselves to be known not null, because NULL on the referenced side cannot match an equijoin comparison.

This is why the proposal introduces BUNC-sets: base-table unique nullable constraint sets. A BUNC-set records uniqueness of the non-null values that can participate in a key join. A BUC-set is the stronger case: a BUNC-set whose columns are also known not null.

This separation keeps the two proof obligations distinct. BUNC-sets supply the PK-side uniqueness proof. Not-nullness is tracked separately, and matters on the FK side when an inner key join would otherwise drop rows whose referencing key is NULL.

7.4.3. How Properties Compose Through Layers

These properties are defined recursively. If view V1 wraps a base table and preserves all required properties, and view V2 wraps V1 and also preserves them, then V2 is valid for use in a key join. The "underlying column" concept (Subclause 7.16, "`<query specification>`") and "counterpart" concept (Subclause 4.27.2, "General rules and definitions") already chain through arbitrary layers of derivation. Row coverage, BUNC-sets, and propagated not-nullness are new, with null-extending joins removing not-null evidence where they can introduce NULLs, but they are defined to compose in the same way.

A schema designer can therefore build a stack of views, each adding projections, joins, or groupings, and as long as each layer preserves the required properties, the top-level view is usable in a key join. The query author sees only the top-level view and treats it as a base table.

7.4.4. A View That Works

The examples that follow use the employees/departments schema introduced in Section 7.2. A schema designer creates a view for use by department managers. The view projects columns but does not filter rows:

```
CREATE VIEW department_directory AS
    SELECT dept_id, name
    FROM departments;
```

A query author can use this view in a key join exactly as if it were the base table:

```
SELECT e.name, dd.name AS department
FROM employees AS e
JOIN department_directory AS dd FOR KEY (dept_id) <- e (dept_id);
```

This works because `department_directory` supplies the PK-side proof facts required by this key join: row coverage and uniqueness.

7.4.5. A CTE That Works

Key joins work through common table expressions in the same way. A query author can define a derived table inline and use it in a key join:

```
WITH
    dept_summary AS (
        SELECT dept_id, name
        FROM departments
    )
SELECT e.name, ds.name AS department
FROM employees AS e
JOIN dept_summary AS ds FOR KEY (dept_id) <- e (dept_id);
```

The same rules apply: `dept_id` in `dept_summary` traces back to `departments.dept_id` via its underlying column, and the DBMS verifies the referential constraint against the base table. The query author could replace the CTE with the `department_directory` view from the previous example, or vice versa, with no change in validity.

Recursive common table expressions (WITH RECURSIVE) are not supported in key joins. The standard already treats recursive CTEs as opaque for BPK/BUC/BUNC-sets and known functional dependencies (Subclause 4.27.16, “Known functional dependencies in a `<query expression>`”), making their properties implementation-defined. Key joins follow the same approach.

7.4.6. LATERAL Derived Tables

A `LATERAL` derived table is a `<table subquery>` that can reference columns from preceding tables in the same FROM clause. Key-join proof facts are not exposed through LATERAL derived tables. Their cardinality and containment properties may depend on the current row of an outer table, so a single operand-surface fact is not enough to prove a key join through them.

7.4.7. Breaking Uniqueness

A view that joins the PK table to another table can duplicate PK rows. Suppose we add a table for projects that are assigned to a department:

```
CREATE TABLE assigned_projects (
    project_id  INTEGER PRIMARY KEY,
    dept_id     INTEGER NOT NULL REFERENCES departments (dept_id),
    name        CHARACTER VARYING(100)
);
```

A reporting view joins departments to their assigned projects:

```
CREATE VIEW department_projects AS
    SELECT d.dept_id, d.name AS dept_name, p.name AS project_name
    FROM departments AS d
    LEFT JOIN assigned_projects AS p FOR KEY (dept_id) -> d (dept_id);
```

The view definition is accepted. The key join inside the view is valid at its own join point because `assigned_projects.dept_id` is a not nullable foreign key. But the output does not have a uniqueness fact for `dept_id`: a department with more than one assigned project appears once per project.

The `employees` table also has a foreign key to `departments.dept_id`. A query that tries to use `department_projects` as the referenced side of that key join is rejected, since the referenced columns are not unique:

```
SELECT e.name, dp.project_name
FROM employees AS e
JOIN department_projects AS dp FOR KEY (dept_id) <- e (dept_id);

-- ERROR:  key join from referencing relation e to referenced relation dp cannot be proven
-- LINE 3: JOIN department_projects AS dp FOR KEY (dept_id) <- e (dept_id);
--                                        ^
-- DETAIL:  Referenced columns dp (dept_id) are not proven unique. A preceding join may
    duplicate rows from referenced relation dp. The relevant operation occurs inside view
    public.department_projects.
```

7.4.8. Restoring Uniqueness with GROUP BY

The repair is to group the referencing table before joining it to `departments`. The grouped derived table has at most one row per `dept_id`, so the subsequent key join cannot duplicate department rows. The LEFT JOIN preserves departments with zero assigned projects, maintaining department row coverage:

```
CREATE VIEW department_project_counts AS
SELECT d.dept_id, COUNT(*) AS project_count
FROM department_projects AS d
GROUP BY d.dept_id;
```

The GROUP BY collapses the duplicated department rows back to one row per `dept_id`. Because `department_projects` preserved all department rows, the grouped view also preserves row coverage for `departments.dept_id`. The grouped `dept_id` column is therefore usable as the referenced side of another key join:

```
SELECT e.name, dpc.project_count
FROM employees AS e
JOIN department_project_counts AS dpc FOR KEY (dept_id) <- e (dept_id);
-- OK: GROUP BY restores uniqueness, LEFT JOIN preserves row coverage
```

Had the aggregate view used a join that did not preserve departments with zero projects, row coverage would be lost. The DBMS would reject the later key join even though uniqueness was satisfied.

7.4.9. Null-extension

An outer join can null-extend an FK column: where the FK column was NOT NULL in the base table, the derived table now has rows where the column is NULL. These null-extended rows cannot match anything in a downstream equijoin, so an inner key join on a null-extended FK column silently drops them and breaks row-preservation of the FK-side derived table.

Consider a schema with regions, stores, and sales:

```
CREATE TABLE regions (
    id   INTEGER PRIMARY KEY,
    name VARCHAR(50)
);
CREATE TABLE stores (
    id        INTEGER PRIMARY KEY,
    name      VARCHAR(50),
    region_id INTEGER NOT NULL REFERENCES regions (id)
);
CREATE TABLE sales (
    id       INTEGER PRIMARY KEY,
    store_id INTEGER NOT NULL REFERENCES stores (id),
    amount   DECIMAL(10,2)
);
```

A reporting view uses LEFT JOINs to list every region together with its sales, keeping regions with no stores and stores with no sales:

```
CREATE VIEW region_sales AS
    SELECT r.id AS region_id, r.name AS region_name,
           sa.store_id, sa.amount
    FROM regions AS r
    LEFT JOIN stores AS s FOR KEY (region_id) -> r (id)
    LEFT JOIN sales AS sa FOR KEY (store_id) -> s (id);
```

`region_sales.store_id` is null-extended for regions with no stores and stores with no sales, even though `sales.store_id` is declared NOT NULL in the base table.

A downstream view cannot use `region_sales` on the FK side of an inner key join to `stores`:

```
CREATE VIEW region_sales_named AS
    SELECT rs.region_name, st.name AS store_name, rs.amount
    FROM region_sales AS rs
    JOIN stores AS st FOR KEY (id) <- rs (store_id);

-- ERROR:  key join from referencing relation rs to referenced relation st cannot be proven
-- LINE 4:     JOIN stores AS st FOR KEY (id) <- rs (store_id);
--                               ^
-- DETAIL:  This inner join could filter rows from rs. Referencing columns rs (store_id)
    can be null because a preceding outer join can null-extend the referencing side. The
    relevant operation occurs inside view public.region_sales.
```

The DBMS rejects the key join at view definition time. The null-extended rows would be dropped by the equijoin, silently omitting regions with no stores and stores with no sales, and the downstream query author would see inflated or deflated totals with no error.

The repair is either to preserve `region_sales` across the downstream key join with an outer join type,

```
CREATE VIEW region_sales_named AS
    SELECT rs.region_name, st.name AS store_name, rs.amount
    FROM region_sales AS rs
    LEFT JOIN stores AS st FOR KEY (id) <- rs (store_id);
```

or to filter the nulls out inside `region_sales` so that `store_id` becomes known not null at the intermediate result.

7.4.10. Nullable UNIQUE Constraints

A referential constraint can reference a UNIQUE constraint whose columns are nullable. Key joins support this. Consider a procurement system where suppliers are registered before their DUNS number (a standard business identifier) is verified:

```
CREATE TABLE suppliers (
    id          INTEGER PRIMARY KEY,
    duns_number CHAR(9) UNIQUE,
    name        CHARACTER VARYING(200),
    region_id   INTEGER REFERENCES regions (id)
);
CREATE TABLE purchase_orders (
    id          INTEGER PRIMARY KEY,
    duns_number CHAR(9) NOT NULL REFERENCES suppliers (duns_number),
    amount      DECIMAL(10,2)
);
```

```
SELECT s.name, po.amount
FROM purchase_orders AS po
JOIN suppliers AS s FOR KEY (duns_number) <- po (duns_number);
```

Suppliers with NULL `duns_number` are unreachable via the equijoin (NULL is not equal to anything), so they do not affect the result. The UNIQUE constraint guarantees that each non-null DUNS number identifies exactly one supplier.

A view that projects columns without filtering also works:

```
CREATE VIEW supplier_directory AS
    SELECT duns_number, name FROM suppliers;

SELECT sd.name, po.amount
FROM purchase_orders AS po
JOIN supplier_directory AS sd FOR KEY (duns_number) <- po (duns_number);
```

The view supplies the PK-side proof facts required by this key join: row coverage and uniqueness. The base table NULLs remain in the view.

Null-extension is a distinct concern from base table nullability. NULLs from the base table are real data (the DUNS number has not been assigned yet); NULLs from outer join null-extension are artifacts that do not represent any row. Both kinds of NULL are inert on the PK side: the equijoin cannot match NULL, so neither a base-table NULL nor a null-extended NULL can participate in a key join result.

The following key join is accepted even though `supplier_by_region` null-extends `duns_number` for regions with no suppliers:

```
CREATE VIEW supplier_by_region AS
    SELECT s.duns_number, r.name AS region_name
    FROM regions AS r
    FULL JOIN suppliers AS s FOR KEY (region_id) -> r (id);

SELECT ps.region_name, po.amount
FROM purchase_orders AS po
JOIN supplier_by_region AS ps FOR KEY (duns_number) <- po (duns_number);
```

Rows of `supplier_by_region` with NULL `duns_number`, whether they came from a base-table NULL or from null-extension, cannot match any `purchase_orders.duns_number`, so they do not affect the result.

7.4.11. Inner Key Joins with Known Not Nullable Referencing Columns

An inner join generally breaks row-preservation: unmatched rows are eliminated. But an inner key join where every FK column is known not nullable is lossless on the referencing side. The constraint guarantees a match for every row, so no rows can be lost.

Using the regions, stores, and sales tables from the null-extension example above, a view that enriches sales with store and region information via two inner key joins preserves row coverage for `sales.id`:

```
CREATE VIEW sale_details AS
    SELECT s.id AS sale_id, st.name AS store_name, r.name AS region_name
    FROM sales AS s
    JOIN stores AS st FOR KEY (id) <- s (store_id)
    JOIN regions AS r FOR KEY (id) <- st (region_id);
```

Each inner key join preserves the relevant referencing rows independently: the FK columns are known not null, and the referential constraint guarantees a match, so no rows are lost. This also preserves the row-coverage fact. The first join preserves row coverage for `sales.id`; the second sees a table that still covers `sales.id` and preserves that fact again. The view is valid as the PK side of a downstream key join.

If any FK column in the chain were nullable, that inner join could eliminate referencing rows, and the row-coverage fact would be lost from that point onward.

7.4.12. Set Operations

Set operations cannot appear on either side of a key join. A set operation produces result columns whose underlying columns come from both operands (Subclause 7.17, "`<query expression>`", Syntax Rule 23d). The key join requirement that each column traces to exactly one underlying base table column rejects this before any other check applies.

Key-join proof facts are not exposed through set operations. A future extension could add specific rules for particular set operations, but the rules here treat them as conservative barriers for key-join proof.

7.4.13. Filtered Views and the PK Side

A key join through a filtered view on the PK side is rejected unless the FK side supplies matching direct key-equality filter evidence; see Section 7.3.2. This differs from a query that filters the base table with a WHERE clause after the join:

```
-- Accepted: join the full base table, then filter the result.
SELECT *
FROM orders AS o
LEFT JOIN customers AS c FOR KEY (id) <- o (customer_id)
WHERE c.active;

CREATE VIEW active_customers AS
    SELECT * FROM customers WHERE active;

-- Rejected: the view is the PK side and lacks row coverage.
SELECT *
FROM orders AS o
LEFT JOIN active_customers AS ac FOR KEY (id) <- o (customer_id);

-- ERROR:  key join from referencing relation o to referenced relation ac cannot be proven
-- LINE 13: LEFT JOIN active_customers AS ac FOR KEY (id) <- o (customer_id);
--                                           ^
-- DETAIL:  Not every o (customer_id) value can be proven to have a matching ac row.
    Referenced relation ac is filtered before this key join. The relevant operation occurs
    inside view public.active_customers.
```

Both queries, if accepted, would produce the same result rows, but would differ from the key join's perspective. In the first query, the join is against the full `customers` table: every order is guaranteed to find its matching customer, and the WHERE clause filters the *result* after the join has been evaluated. In the second query, `active_customers` is the PK side, and it is missing inactive customers. An order that references an inactive customer will not find a match.

This asymmetry is inherent in row coverage. A key join guarantees that the join will behave as a proper foreign key traversal, producing exactly one match for each FK row. A filtered PK side breaks this guarantee unless the missing PK values are matched by filters that remove the corresponding FK rows before the join.

7.4.14. Safe View Composition

A schema designer can build layered helper views that pre-join a central table to its related tables. Query authors then write against these views instead of repeating the joins in every query.

Without key joins, this layering is fragile. A schema designer who builds a view on top of another view must manually verify that the underlying view preserves the right properties. If the underlying view changes, a column that behaved like a primary key may start producing duplicates or losing rows, and every dependent view silently returns wrong results. With key joins, the DBMS verifies these properties automatically at view definition time and blocks schema changes that would break them.

The example uses an order-processing schema with a central `orders` table referencing `customers`, `order_statuses`, `shipping_methods`, `payment_methods`, `currencies`, `order_priorities`, `tax_codes`, and `discount_types`; `customers` references `countries` and `customer_types`; and `order_items` references `orders` and `products`. A schema designer creates three layered views. The first enriches customers with their country and type:

```
CREATE VIEW customer_details AS
    SELECT c.id AS customer_id, c.name, co.country_name,
           ct.type_name AS customer_type
    FROM customers          AS c
    JOIN countries          AS co FOR KEY (id) <- c (country_id)
    JOIN customer_types     AS ct FOR KEY (id) <- c (customer_type_id);
```

The second enriches orders with customer details and lookup tables:

```
CREATE VIEW order_details AS
    SELECT o.id AS order_id, o.order_date,
           cd.name AS customer, cd.country_name, cd.customer_type,
           os.status_name, sm.method_name AS shipping,
           pm.method_name AS payment, cur.currency_code,
           op.priority_name, tc.code_name AS tax_code,
           dt.type_name AS discount
    FROM orders              AS o
    JOIN customer_details    AS cd  FOR KEY (customer_id) <- o (customer_id)
    JOIN order_statuses      AS os  FOR KEY (id)  <- o (order_status_id)
    JOIN shipping_methods    AS sm  FOR KEY (id)  <- o (shipping_method_id)
    JOIN payment_methods     AS pm  FOR KEY (id)  <- o (payment_method_id)
    JOIN currencies          AS cur FOR KEY (id)  <- o (currency_id)
    JOIN order_priorities    AS op  FOR KEY (id)  <- o (order_priority_id)
    JOIN tax_codes           AS tc  FOR KEY (id)  <- o (tax_code_id)
    LEFT JOIN discount_types AS dt  FOR KEY (id)  <- o (discount_type_id);
```

The key join on `customer_details` guarantees that each order's `customer_id` matches exactly one row in `customer_details`. No order rows are lost and no order rows are duplicated, so `order_details` contains exactly one row per order. A schema designer can safely build further views on top of it (e.g., an `orders_by_country` summary) with the same guarantee.

The third view enriches order items with order details and product information:

```
CREATE VIEW order_item_details AS
    SELECT oi.id AS item_id, oi.amount,
           od.order_date, od.customer, od.country_name,
           p.name AS product_name
    FROM order_items    AS oi
    JOIN order_details  AS od FOR KEY (order_id) <- oi (order_id)
    JOIN products       AS p  FOR KEY (id)       <- oi (product_id);
```

A query author who needs customer information writes against `customer_details`. One who needs order-level data uses `order_details`. One who needs line-item data uses `order_item_details`. If a schema change would break a property that a downstream view depends on, the DBMS rejects the change and alerts the schema designer to the problem.

7.5. Constraint Requirements

A key join requires each proof source to be ENFORCED and NOT DEFERRABLE. This applies to the referential constraint, to the primary key or unique constraint used to prove PK-side uniqueness, and to any NOT NULL constraint used to prove that an inner key join cannot drop rows with null referencing key values.

The compile-time guarantees of a key join depend on these constraints actually being maintained. A NOT ENFORCED constraint provides no data integrity guarantee; the data may violate the constraint at any time. A DEFERRABLE constraint permits checking to be postponed within a transaction. A key join evaluated while such a postponement is in effect could produce wrong results

even though the constraint will eventually be checked.

If a key join depends on a constraint that is NOT ENFORCED or DEFERRABLE, the query is rejected at compile time. If the query compiles, the join is valid under the declarations used by the proof.

7.5.1. Exclusion of Temporal Referential Constraints

A key join does not reference a referential constraint that includes a `<referencing period specification>` (i.e., a temporal foreign key as defined in Feature T181, "Application-time period tables"). Temporal referential constraints have range-containment semantics: the FK row's application-time period must be a subset of the union of matching PK rows' periods. A single FK row can match multiple PK rows whose periods collectively cover it, breaking the 1:1 cardinality guarantee. Supporting temporal constraints would require purpose-built syntax beyond the scope of this proposal; the exclusion is forward-compatible with a future extension.

7.6. Schema Changes and Dependent Objects

A stored definition that contains a key join is dependent on every schema object used to establish the proof. This includes the referential constraint, supporting primary key or unique constraints, consumed NOT NULL constraints, view definitions that expose proof facts, and routines used in proof predicates. This uses the same dependency mechanisms already defined in the standard, including the rules for dropping constraints (Subclause 11.26, "`<drop table constraint definition>`").

Stored definitions include views and SQL functions with stored SQL bodies. A view that merely exposes traceable columns need not record those proof-source dependencies until a stored key join consumes that view as a proof surface.

The consequences are:

- **RESTRICT**: Dropping a proof source that a stored key join depends on is prohibited. The DROP fails.
- **CASCADE**: Dropping the proof source causes all dependent stored definitions to be dropped or otherwise invalidated, recursively.

If a constraint that a stored key join depends on is altered so that it is NOT ENFORCED or becomes DEFERRABLE, the alteration must be blocked unless dependent objects are handled by the applicable cascading operation. The constraint no longer satisfies the conditions under which the key join was defined.

The `VIEW_CONSTRAINT_USAGE` view added by the Schemata part of this proposal makes the table-constraint subset of these dependencies visible to schema designers. For a view containing a key join, this includes the referential constraint, the supporting primary key or unique constraint, and any NOT NULL constraints consumed as proof sources. Routine and user-defined ordering dependencies remain handled by the existing dependency mechanisms for those objects.

Prepared statements containing a key join are compiled against the current schema. If any proof source used by the key join is dropped or altered before the statement is executed, the statement is re-prepared or fails, following the existing rules for prepared statement invalidation.

7.6.1. Real-World Consequences

The preceding examples were simplified to illustrate individual properties. The scenarios that follow show how the same mechanisms prevent silent data corruption in more realistic settings. Both use the following schema, where each customer has one address:

```
CREATE TABLE addresses (
    id    INTEGER PRIMARY KEY,
    street CHARACTER VARYING(200),
    city   CHARACTER VARYING(100)
);

CREATE TABLE customers (
    id         INTEGER PRIMARY KEY,
    name       CHARACTER VARYING(100),
    address_id INTEGER NOT NULL REFERENCES addresses (id),
    active     BOOLEAN DEFAULT TRUE NOT NULL
);

CREATE TABLE orders (
    id          INTEGER PRIMARY KEY,
    amount      DECIMAL(10,2),
    customer_id INTEGER NOT NULL REFERENCES customers (id)
);
```

A reporting view joins customers to their addresses and is used as the PK side of a key join in a downstream view:

```
CREATE VIEW customer_details AS
    SELECT c.id AS customer_id, c.name, a.city
    FROM customers AS c
    JOIN addresses AS a FOR KEY (id) <- c (address_id);

CREATE VIEW revenue_by_city AS
    SELECT cd.city, SUM(o.amount) AS total
    FROM orders AS o
    JOIN customer_details AS cd FOR KEY (customer_id) <- o (customer_id)
    GROUP BY cd.city;
```

This works because `customer_details` supplies the PK-side proof facts required for the `customer_id` key: row coverage and uniqueness. The next two sub-subsections show how the dependency mechanism reacts when each fact is broken in turn: the first by a schema migration that breaks uniqueness, the second by a data maintenance change that breaks row coverage.

7.6.2. Schema Migration Breaking Uniqueness

A later migration adds support for multiple addresses per customer. The schema designer moves the foreign key from `customers` to `addresses` and drops the existing views (via CASCADE) to redefine them:

```
DROP VIEW customer_details CASCADE;

ALTER TABLE addresses ADD COLUMN customer_id INTEGER;

UPDATE addresses
SET customer_id = (
    SELECT c.id
    FROM customers AS c
    WHERE c.address_id = addresses.id
);

ALTER TABLE addresses ALTER COLUMN customer_id SET NOT NULL;
ALTER TABLE addresses ADD FOREIGN KEY (customer_id) REFERENCES customers (id);
ALTER TABLE customers DROP COLUMN address_id;

CREATE VIEW customer_details AS
    SELECT c.id AS customer_id, c.name, a.city
    FROM customers AS c
    LEFT JOIN addresses AS a FOR KEY (customer_id) -> c (id);
```

The recreated `customer_details` view is accepted. The LEFT JOIN preserves customer row coverage, but `customer_id` is no longer unique in the view output: a customer with multiple addresses appears once per address. When the schema designer attempts to recreate the downstream revenue view, the DBMS rejects it:

```
CREATE VIEW revenue_by_city AS
    SELECT cd.city, SUM(o.amount) AS total
    FROM orders AS o
    JOIN customer_details AS cd FOR KEY (customer_id) <- o (customer_id)
    GROUP BY cd.city;

-- ERROR:  key join from referencing relation o to referenced relation cd cannot be proven
-- LINE 4:     JOIN customer_details AS cd FOR KEY (customer_id) <- o (customer_id)
--                                          ^
-- DETAIL:  Referenced columns cd (customer_id) are not proven unique. A preceding join may
     duplicate rows from referenced relation cd. The relevant operation occurs inside view
    public.customer_details.
```

Without key joins, the join would silently count each order once per address, inflating revenue figures, and the inflated totals would go undetected until someone noticed the numbers were wrong.

7.6.3. Data Maintenance Breaking Row-Preservation

Privacy regulations require anonymization of inactive users. The schema designer modifies the `customers` table to allow NULL addresses:

```
ALTER TABLE customers ALTER COLUMN address_id DROP NOT NULL;

-- ERROR:  cannot drop constraint customers_address_id_not_null on table customers because
    other objects depend on it
-- DETAIL:  view customer_details depends on constraint customers_address_id_not_null on
    table customers
-- HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

The `customer_details` view uses an inner key join from `customers` to `addresses`, with `address_id` as the FK column. That stored key-join proof depends directly on the NOT NULL constraint on `address_id`: without it, the inner join could drop customers whose address is unknown. The `DROP NOT NULL` command is rejected while `customer_details` depends on that proof. The schema designer must explicitly drop or replace the view before allowing nullable addresses.

The schema designer then recreates `customer_details` with a LEFT JOIN, preserving anonymized customers with NULL address fields:

```
DROP VIEW customer_details;
CREATE VIEW customer_details AS
SELECT c.id AS customer_id, c.name, a.city
FROM customers AS c
LEFT JOIN addresses AS a FOR KEY (id) <- c (address_id);
```

Without key joins, the schema designer could drop the NOT NULL constraint, run a nightly anonymization job that sets `address_id` to NULL, and never notice that the inner join in the original `customer_details` silently excludes those customers. A downstream `revenue_by_city` report over `customer_details` would then lose orders for anonymized customers, producing deflated totals with no error.

7.6.4. Error Messages and Privilege Scoping

When a key join fails because the required referential constraint does not exist, or because a derived table does not satisfy the required properties, the DBMS raises an error condition. That error condition may populate diagnostic area fields such as TABLE_NAME, COLUMN_NAME, and CONSTRAINT_NAME with identifiers of the underlying base tables and constraints.

If the query author does not hold privileges on those underlying objects, revealing their names would be an information leak. The standard already addresses this: the diagnostic area suppression rules in Subclause 23.1, "`<get diagnostics statement>`" require that TABLE_NAME, COLUMN_NAME, and constraint identifiers be replaced with zero-length character strings when no relevant privilege is held by the current authorization identifiers. Key join errors are subject to these existing rules.

In practice, a query author with privileges on the underlying tables might see an error identifying the missing constraint. A query author who accesses the same tables only through views would see a generic error such as "key join cannot be satisfied," with no object names disclosed.

7.7. Soundness of the Specification

The Specification is sound because it is accept-by-proof: the DBMS accepts a key join only when recognized proof rules establish all three conditions of the Definition against the intermediate result at the point the join is evaluated. Unknown or opaque constructs do not need special cases; by default they expose no row-coverage, uniqueness, foreign-key containment, or not-nullness facts that can discharge a key-join obligation, so no proof is formed. The propagation rules of Section 7.4 carry established facts through derived tables compositionally, so the same argument applies whether the operands are base tables or derived tables.

The Specification is not complete. Some queries that would in fact satisfy the Definition for a given database state are still rejected, because the proof obligations are discharged from schema declarations and query structure alone, not from inspection of the data. This is intentional: a sound-but-incomplete validator rejects some queries for which a stronger proof might exist, but never accepts a key join without the proof facts required by the Definition.

A DBMS may also require the compared key positions to have the same declared type, collation, and equality semantics for proof purposes. The mathematical definition only requires an equijoin comparison, but the compile-time proof must be able to determine from declarations and query structure that the relevant uniqueness and row-coverage facts, and the join comparisons, use the same equality semantics. If that cannot be established, the key join is rejected.

8. Design Rationale

8.1. Compile-Time Verification

A key join could be verified either at compile time (during query preparation) or at runtime (during query execution). This proposal requires compile-time verification.

8.1.1. The Query Author's Contract

A key join is a declaration: "this join follows a foreign key." Compile-time verification provides a hard guarantee: if the query prepares successfully, the referential constraint exists and the join is semantically valid. This does not depend on the contents of the tables.

8.1.2. Runtime Checks Are Insufficient

Runtime verification has several problems.

Errors depend on data, not on validity. A query that joins on the wrong columns may produce correct-looking results for months, as long as the data does not expose the mismatch. The hotel scenario in Section 7.3.4 is exactly this case: the query worked in testing because the test dataset had only one hotel. A runtime check would only catch the error when problematic data actually flows through the join, which may be long after deployment.

Errors are intermittent and hard to reproduce. If a runtime check fires only when certain rows are present, the error may appear in production but not in development or staging environments.

Failures may arrive only after substantial work. For long-running analytical queries, a runtime check may not encounter the offending row until the query has scanned large tables, consumed significant resources, or produced part of a result. The failure is then late and operationally expensive. Compile-time verification rejects the invalid key join before execution begins.

Deferred checks cannot prevent wrong results. By the time a runtime check fires, the query is already executing. Wrong results may have already been returned to the application, written to other tables, or used in downstream computations.

Runtime checks cannot validate stored definitions. A prepared statement or view definition is compiled once and executed many times, potentially against changing schemas. Compile-time verification catches problems when the statement is prepared or the view is created.

8.1.3. Key Joins Aid Testing

Constructing test data that exercises every join path in a complex query is difficult. The hotel scenario is instructive: the join on `room_number` alone is wrong, but a test dataset needs multiple hotels

with overlapping room numbers to reveal the bug. A tester who does not independently verify the foreign key structure will not think to construct that specific dataset. The bug hides not because the tester is unskilled, but because the test data must be specifically crafted to expose a structural error that has nothing to do with the query's business logic.

Key joins eliminate this class of structural errors at compile time. This frees the tester to focus on verifying that the query produces correct results for the intended business logic, with confidence that the joins are structurally sound.

8.1.4. Compile-Time Verification Enables Composition

The derived table support described earlier relies on compile-time analysis of view definitions to determine whether referential properties are preserved through layers of derivation. This analysis examines the structure of the query, not the data. A runtime approach could not provide these guarantees without executing the view's query first.

The separation of duties between schema designer and query author also depends on compile-time verification. The schema designer constructs views that preserve the required referential properties; the query author uses them. This contract works only if the DBMS can verify the view's properties when the query is compiled, not when it happens to be run with particular data.

8.2. Why Arrow Syntax

The authors originally considered `TO` and `FROM` as keywords for indicating direction, but `FROM` in a join clause causes confusion with the `FROM` clause itself. Testing with focus groups confirmed that arrows are clearer and less ambiguous. The choice also resembles SQL:2023 SQL/PGQ, whose graph pattern syntax likewise uses ASCII arrows.[3]

8.3. Why Arrow Direction Is Fixed by the FK, Not Chosen by the Author

The fixed arrow direction realises the local-determinability part of the Intention. The arrow direction in a key join encodes the side of the newly introduced table. `<-` means the newly introduced table is the PK side; `->` means it is the FK side. The reader tells the side from the arrow alone, without reading the aliases on either side.

An alternative design would let the query author place either column list first and flip the arrow to match, so the same key join could be written either way. Under that alternative, the arrow no longer fixes the side of the newly introduced table.

The fixed convention keeps the all-`<-` scan described in Section 8.9 a single pass. A reader can tell at a glance whether every join uses `<-`, and therefore whether the first table in the FROM clause is preserved end-to-end. Under the alternative, the reader would have to parse each key join individually, reading the arrow and checking which alias sits on which side, to decide whether that key join preserves or duplicates the running row set.

8.4. Why Column Lists Instead of Constraint Names

An alternative design would name the referential constraint directly (e.g., `JOIN ... FOR KEY fk_books_author`). This has two problems. First, when the same base table appears multiple times in a query, the constraint name alone cannot distinguish which instance is intended. Second, referential constraints are defined between base tables; derived tables do not have constraint names. Since key joins are designed to work through views and CTEs, the syntax must use column lists, which exist at every level of derivation. Constraint names do not.

[3] Peter Eisentraut, "SQL:2023 is finished: Here is what's new", 2023-04-04, https://peter.eisentraut.org/blog/2023/04/04/sql-2023-is-finished-here-is-whats-new.

8.5. Why the Joined Table's Column List Takes No Alias

A key join's syntax is `JOIN T AS a FOR KEY (cols) <- b (cols)`: the column list immediately after `FOR KEY` carries no alias, while the column list after the arrow is preceded by one.

The right operand of a JOIN clause is almost always a single named base table, view, CTE, or subquery, so the columns listed after `FOR KEY` belong unambiguously to that operand. Adding an alias would be redundant and visually noisy. The other side of the arrow is different: it names columns from a table already in scope, and in a query with many joins that table must be disambiguated with an alias.

The remaining edge case is a parenthesized join on the right of `JOIN`. Even there, an alias before the first column list is needed only when several newly joined tables expose referred columns with the same name. We do not expect that case to be common enough to justify adding the alias position to ordinary key joins today. The syntax leaves room for it: a future revision could add an optional alias before the first column list if usage shows that the edge case matters in practice.

8.6. Why Explicit Columns Instead of Inference

An inference-based design, where the DBMS determines the join columns automatically from the constraint, has been tried in practice. SQL Anywhere (Sybase, in releases since the late 1990s) provides a `KEY JOIN` syntax that infers the foreign key relationship without requiring the query author to name any columns:

```
SELECT rl.id, rl.amount * c.rate AS usd_amount, r.receipt_date
FROM receipt_lines AS rl
KEY JOIN receipts AS r
KEY JOIN currencies AS c;
```

This works until the schema evolves. If a `currency` column with a foreign key to `currencies` is later added to `receipts`, the KEY JOIN with `currencies` silently re-resolves to the new foreign key, joining through `receipts` instead of through `receipt_lines`. The query returns different results with no error.

Explicit column lists prevent this: the query author declares which columns the join follows, and schema changes that invalidate the declaration produce an error rather than silently changing the query's meaning.

8.7. Indirect Foreign Keys Are Not Supported

A key join follows a single referential constraint. It does not support transitive ("indirect") foreign key paths, even when the intermediate column is the same column. For example, `time_cards.person_id` can reference `employees.person_id`. The same `employees.person_id` column can also be a foreign key to `people.person_id`. A query author cannot key join `time_cards` directly to `people`. The `FK -> PK = FK -> PK` path must still be explicit:

```
SELECT t.work_date, p.name
FROM time_cards AS t
JOIN employees AS e FOR KEY (person_id) <- t (person_id)
JOIN people AS p FOR KEY (person_id) <- e (person_id);
```

An indirect key join would require the DBMS to compose referential constraints across intermediate tables, introducing ambiguity (multiple paths may exist), hiding the intermediate table from the query, and preventing independent verification of each join. Requiring explicit joins through each table keeps the query's intent visible.

8.8. Benefits

The preceding subsections have justified individual design choices. The benefits below summarise what those choices buy in aggregate.

- **Queries document their intent.** A key join states which referential constraint the join follows. The direction of the arrow shows which is the FK table and which is the PK table.
- **Schema changes surface as errors, not as wrong results.** A stored definition that contains a key join is dependent on the proof sources used to validate that key join, including the referential constraint, supporting uniqueness constraints, consumed NOT NULL constraints, proof-exposing view definitions, and routines used in proof predicates. Stored definitions include views and SQL functions with stored SQL bodies. Dropping any of these with RESTRICT is blocked; dropping with CASCADE drops or invalidates the dependent stored definitions. This uses the same dependency mechanism already defined in the standard.
- **Views compose safely.** A schema designer can build layered views that pre-join related tables. Key joins verify the required properties at each layer, and dependency tracking prevents schema changes from silently breaking downstream views (see Section 7.4.14).
- **Missing join columns are caught.** Composite foreign keys are a common source of bugs: developers forget one column and the join looks correct in simple test data. A key join catches this immediately, as shown in the hotel booking example in Section 7.3.4.
- **Fan traps are caught at compile time.** Joining a referenced table to two of its referencing tables silently multiplies rows and inflates aggregates in traditional equijoins. A key join rejects this pattern at compile time (Section 7.3.1).
- **No same-name requirement.** Unlike USING, key joins work when FK and PK columns have different names (e.g., `orders.customer_id` referencing `customers.id`). And unlike NATURAL or USING, key joins are unambiguous when multiple tables in the query share column names.
- **Immune to column additions.** A USING clause can break when an unrelated schema change adds a column with the same name to another table in the join. Key joins name the specific table reference for each column, so they are immune to this problem.

8.9. Readability at Scale

The local-readability part of the Intention shows up most concretely in larger queries. Consider a typical order-processing query that joins a central `orders` table to ten related tables. The query author wrote the joins in the order they came to mind, arbitrarily starting the FROM clause with `customers`:

```
SELECT o.id, o.order_date, oi.amount,
       c.name AS customer, co.country_name,
       os.status_name, sm.method_name AS shipping,
       pm.method_name AS payment, cur.currency_code,
       op.priority_name, tc.code_name AS tax_code,
       dt.type_name AS discount
FROM customers           AS c
JOIN countries           AS co  ON co.id  = c.country_id
JOIN customer_types      AS ct  ON ct.id  = c.customer_type_id
JOIN orders              AS o   ON c.id   = o.customer_id
JOIN order_statuses      AS os  ON os.id  = o.order_status_id
LEFT JOIN discount_types AS dt  ON dt.id  = o.discount_type_id
JOIN shipping_methods    AS sm  ON sm.id  = o.shipping_method_id
JOIN payment_methods     AS pm  ON pm.id  = o.payment_method_id
JOIN order_items         AS oi  ON o.id   = oi.order_id
JOIN currencies          AS cur ON cur.id = o.currency_id
JOIN order_priorities    AS op  ON op.id  = o.order_priority_id
JOIN tax_codes           AS tc  ON tc.id  = o.tax_code_id
WHERE o.id = 100;
```

Every ON clause looks the same: two columns compared for equality. Nothing in the syntax reveals which table references which, or whether the join follows a foreign key at all. The reader must cross-reference the schema to verify each condition.

The first refactoring step is mechanical: rewrite each ON clause as a FOR KEY clause, leaving the join order untouched. The arrow direction is fixed by the schema, not chosen by the query author (see Section 8.3).

```
SELECT o.id, o.order_date, oi.amount,
       c.name AS customer, co.country_name,
       os.status_name, sm.method_name AS shipping,
       pm.method_name AS payment, cur.currency_code,
       op.priority_name, tc.code_name AS tax_code,
       dt.type_name AS discount
FROM customers           AS c
JOIN countries           AS co  FOR KEY (id)          <- c (country_id)
JOIN customer_types      AS ct  FOR KEY (id)          <- c (customer_type_id)
JOIN orders              AS o   FOR KEY (customer_id) -> c (id)
JOIN order_statuses      AS os  FOR KEY (id)          <- o (order_status_id)
LEFT JOIN discount_types AS dt  FOR KEY (id)          <- o (discount_type_id)
JOIN shipping_methods    AS sm  FOR KEY (id)          <- o (shipping_method_id)
JOIN payment_methods     AS pm  FOR KEY (id)          <- o (payment_method_id)
JOIN order_items         AS oi  FOR KEY (order_id)    -> o (id)
JOIN currencies          AS cur FOR KEY (id)          <- o (currency_id)
JOIN order_priorities    AS op  FOR KEY (id)          <- o (order_priority_id)
JOIN tax_codes           AS tc  FOR KEY (id)          <- o (tax_code_id)
WHERE o.id = 100;
```

Both arrow directions appear. Nine joins use `<-`: the joined table is the PK side of a lookup from a table already in scope. Two use `->`: `orders` is reached from `customers`, and `order_items` is reached from `orders`. Each `->` is a point where the query's row count can grow, the mechanism that makes fan traps possible (Section 7.3.1). Key join validation tracks this against the intermediate result.

The second refactoring step is optional. A query author who wants the result to map one-to-one to a specific table's rows can place that table first in the FROM clause and rearrange the joins so every arrow becomes `<-`. Reaching this shape requires a root table in the foreign key graph (one that no other table references in the query); only a root can sit at the start with every other table joined as the PK side of a `<-`. Placing `order_items` first:

```
SELECT o.id, o.order_date, oi.amount,
       c.name AS customer, co.country_name,
       os.status_name, sm.method_name AS shipping,
       pm.method_name AS payment, cur.currency_code,
       op.priority_name, tc.code_name AS tax_code,
       dt.type_name AS discount
FROM order_items         AS oi
JOIN orders              AS o   FOR KEY (id) <- oi (order_id)
JOIN customers           AS c   FOR KEY (id) <- o (customer_id)
JOIN countries           AS co  FOR KEY (id) <- c (country_id)
JOIN customer_types      AS ct  FOR KEY (id) <- c (customer_type_id)
JOIN order_statuses      AS os  FOR KEY (id) <- o (order_status_id)
JOIN shipping_methods    AS sm  FOR KEY (id) <- o (shipping_method_id)
JOIN payment_methods     AS pm  FOR KEY (id) <- o (payment_method_id)
JOIN currencies          AS cur FOR KEY (id) <- o (currency_id)
JOIN order_priorities    AS op  FOR KEY (id) <- o (order_priority_id)
JOIN tax_codes           AS tc  FOR KEY (id) <- o (tax_code_id)
LEFT JOIN discount_types AS dt  FOR KEY (id) <- o (discount_type_id)
WHERE o.id = 100;
```

Every join uses `<-`: each newly introduced table is the referenced (PK) side, guaranteed to be preserved by the key join. No join can duplicate `order_items` rows. The result maps one-to-one to a subset of the `order_items` rows (a subset because the WHERE clause filters). This guarantee is structural, verified at compile time, and requires no runtime checks.

9. Conclusion

This paper makes three contributions:

- **Semantic:** key joins name the common pattern of following a referential constraint while preserving the referencing rows and enriching each all-non-null referencing key with its unique referenced row. The Definition in Section 6 captures this intention as three multiset conditions.
- **Cognitive:** key joins make the referencing and referenced sides, and the compared columns, local to the query text. The fixed arrow direction lets readers follow chains of key joins without reconstructing the foreign-key relationship from the schema.
- **Operational:** key joins are checked at compile time from declarations and query structure. The proof rules are sound rather than complete, and propagate through views, CTEs, and subqueries where the required facts can be established.

Key joins add a declarative, compile-time-checked mechanism for a common kind of SQL join. They surface errors early, document query intent, and build on existing standard concepts, extending them only where new behaviour is needed. We encourage ISO/IEC JTC 1/SC 32/WG 3 to consider integrating key joins into the SQL standard.

end of paper