Reduction of N1QL v4 to SQL++ Core

Kian Win Ong, Yannis Papakonstantinou, Gerald Sangudi

We have already seen in Table 1 the subset of SQL++ supported by N1QL, the query language of the Couchbase JSON database. Rather than allowing the users to write arbitrary SQL++, with the risk of allowing the user to write an unsupported query, N1QL is essentially a dialect of SQL++ that guides the user towards expressing only the supported subset. For example, while SQL++ allows arbitrary joins, N1QL allows only efficient joins - that is, joins between primary keys and references. In another example of a restriction, N1QL rather than allowing arbitrary subqueries that would range over attribute/value pairs or array elements, it introduces special syntactic constructs that are specialized to range over just attribute/value pairs or just array elements of nested arrays.

In this section we explain how the special syntactic constructs of N1QL’s native syntax (N1QL version 4) are formally explained via a reduction to SQL++ core, i.e., can be seen as syntactic sugar over the SQL++ core. The following discussion is limited to N1QL features pertaining to the SELECT and FROM functionality of SQL++. Occasionally, we reduce N1QL to SQL (rather than SQL++ core). In such case, the further reduction to SQL++ core is identical to SQL’s reduction to SQL++ core.

The $n1$from-term corresponds to the from-item. Similarly to the definition of from-item, line 8 provides the base of the induction and lines 9-11 provide the inductive step. Unlike from_item that allows an arbitrary collection expression to produce bindings, the n1from-term expects a path to provide the collection.

The n1use-keys-clause (line 8) restricts the bindings delivered by the n1from-term. The following rewriting reduces n1use-keys-clause into a SQL++ core expression. In order to emulate the function of keys in SQL++, we assume that the collection expression $e$ returns tuples, which are bound to $v$ and have a designed primary key attribute $p$.

\[
\begin{align*}
  e & \text{ AS } v \text{ USE PRIMARY KEYS } k \Rightarrow \\
  (\text{FROM } e \text{ AS } v) & \text{ WHERE (SOME } r \text{ IN } k \text{ SATISFIES } v.p = r) \\
  & \text{ SELECT ELEMENT } v
\end{align*}
\]

N1QL’s JOIN construct (lines 9 and 15) has introduced the special n1on-keys-clause in lieu of SQL’s arbitrary JOIN condition (lines 9 and 10 of Figure 1), because the n1on-keys-clause allows the user to express only foreign-key-to-primary-key joins, which are generally considered to be efficient joins. Therefore the n1on-keys-clause is easily reduced to SQL by the following reduction. Assume that the left n1from-term (line 9) $t_l$ defines an alias variable $v_l$ (possibly among others) that binds to tuples that have a primary key attribute $p$. (Again, in N1QL’s case the primary key attribute would be implicit rather than explicit.)

\[
\begin{align*}
  t_l & \text{ n1join-type } JOIN r \text{ AS } r_v \text{ ON KEYS } e(r_v) \Rightarrow \\
  t_l & \text{ n1join-type } JOIN r \text{ AS } r_v \text{ ON } \\
  & \text{ SOME } x \text{ IN } e(r_v) \text{ SATISFIES } x = v_l.p
\end{align*}
\]

The reduction of N1QL’s FLATTEN to SQL++ core was already discussed in Section 5.1.
Figure 19: The reduced N1QL subset