Preamble

This document describes the Intellectual Property to lies at the core of each Edge Intelligence server and which enables each edge server to run autonomously while providing excellent performance against large data volumes without prior design.

Hardware Efficiency

Modern hardware has key performance characteristics that the Intellectual Property aims to both exploit and mitigate. Firstly, data transfers from storage to memory and from memory to CPU cache have fast transfer rates while the latency to initiate each transfer is relatively long compared to the length of a CPU cycle.

The primary goal of the Intellectual Property is to maximise the amount of relevant data contained within each transfer and minimise the overall number of transfers required to service a query. This allows for larger transfer sizes which can fully exploit fast transfer rates and also reduce the effective overhead of transfer latency. Good spatial locality optimises the amount of relevant data contained within a transfer.

Modern CPUs, operating systems and file systems routinely attempt to mitigate latency through read-ahead strategies which speculatively assume that data which immediately follows a transfer along the data address dimension will be similarly relevant. Data with poor spatial locality invalidates that assumption; while data with good spatial locality assists a read-ahead strategy.

The trend in contemporary hardware has been to increase the number of CPU cores rather than increase CPU speed – partially because increasing CPU speeds largely makes them more inefficient because of the time spent waiting during transfer latencies. Therefore, in addition to good spatial locality, data structures also need to be predominantly linear and homogenous to be able to exploit multiple core architectures.

Now, contrast the requirements of modern hardware with a conventional hierarchical index structure, such as a B-Tree index, which typically exhibit poor spatial-locality. For example, a block within a B-Tree may contain thousands of key entries, yet only one of those entries may be relevant to the query at hand and therefore the block transfer, and its attendant latency cost, typically returns less than 1% of useful data. Moreover, any read-ahead attempt is very unlikely to return the next block actually required to continue navigation through the tree structure and therefore this inefficiency is repeated at each level of the tree visited – even if it is cached in memory. Further still, a hierarchical index structure will defeat any attempt to employ multiple cores during a single navigation because the nature of a tree algorithm is necessarily sequential. The storage structures defined in the Intellectual Property avoid the inefficiencies of hierarchical indexing experienced during both index writes and reads.

Access Model

The Intellectual Property is designed to be universal and independent of any particular data model such as the relational model and it defines a generic access model which is functionally complete for the relational model, but does not assume it. The access model provides a fundamental set of operators which function on collections of scalar elements and supports operations such adding and deleting a collection instance; adding and deleting an element to an existing collection instance;

retrieving collection instances or element aggregates by arbitrary element predicate expressions; as well as operations for managing transaction state.

The concept of a collection is universal to virtually all data models whether they be tuples, vectors, objects or documents. However, the internal structure of a real-world collection can vary enormously, while the access model in the Intellectual Property only provides for a collection which is a structure-free bag of scalar elements. The arbitrary structure of a collection is resolved by decomposing each structure into its scalar components and serialising the structure into the metadata associated with each scalar element. For example, a geometric position P with a coordinate pair (X,Y) may decompose into two scalar numbers with meta paths P/X and P/Y.

The access model provides the concept of intent which recognises how a scalar element is used rather than its data type. For example, an element may be involved in an equivalence expression regardless of whether it is a Boolean, number, time or textual value; whereas a textual string can be tested for the presence of a contained property such as a word while a number data type cannot because it has to be interpreted in whole.

An element may have multiple intents and each intent determines which operators within the access model are applicable to it. Part of the Intellectual Property is concerned with the mapping from a relational schema to collections, elements, intents and operators.

Storage Structures

Each intent within the access model has an associated family of fragments which provide optimal spatial locality for the operators associated with that intent. In each fragment family there will be many thousands of fragments per intent per element; whereby each fragment supports a specific operator and operand subset and each element instance will be denoted in multiple fragments. Therefore for a single collection path, there will typically be many millions of individual fragments supporting all combinations of operators and operands against all element paths.

Any data stored in a fragment is typically only part of an element but is always just sufficient to support the operator associated with the fragment family. Indeed, while a fragment will always guarantee 100% recall (will return all relevant instances) it often relaxes precision such that a superset of the relevant data may be returned. However, by combining the results from other fragment families and/or filtering of the result set, the final result is always 100% precise. The relaxation of precision allows for better spatial locality, more efficient compression and greater throughput to provide a more efficient fragment structure overall.

Fragments are compressed using fragment family specific RLE methods which allow fragment data to be interpreted as read, avoiding any prior inflation, to optimise CPU cache efficiencies.

In all cases, fragments are homogenous within their own family and are linear append-only structures. Updates are handled by tombstoning an element instance and appending a new version, therefore an element instance may appear multiple times with a single fragment. For this reason, fragments are read in reverse time order to facilitate version resolution.

Every fragment is divided into common time-frame boundaries that permit query correlation across fragments and fragment families, allow query navigation to switch between fragments as appropriate and also to permit multiple cores to read a single fragment without contention. Statistics are also maintained about every frame and super set of frames for every fragment family to permit the rapid elimination of frames irrelevant to any particular operand set.

The database is divided into transactional heaps of active transactions and fragments of committed data – where a heap is simply a journal of transactions with transactional content. Committed heap data is migrated to fragment structures using multiple cores when there is sufficient heap data to amortize the cost of appending to multiple fragments; and heaps are dropped once they have been successfully migrated. Queries will read heaps first and resolve any transaction context prior to traversing fragments containing committed data.

A commit operation simply requires pending transaction data to be appended to an active heap and it can participate in a group commit cycle to minimise the number of write transfers, minimise storage synchronisation time and optimise commit throughput.

Joins

The access model supports equi-joins, semi-joins and cross-joins through a bind operator. This operator is able to exploit a fragment family which effectively provides pre-calculated hash buckets for each side of a hash-join operation. The knowledge of which hash buckets are relevant to each side of the join allow hash buckets to be immediately eliminated and enables narrow joins to perform like nested-loop joins; while wide-joins perform much more efficiently than conventional hash-joins because the hash buckets have always been pre-computed.