Chapter 9

The approXQL Query Engine

The approXQL query engine is a prototypical implementation of the theoretical concepts introduced throughout this thesis. Each of the three query-evaluation methods proposed in Chapters 6 and 7 has been implemented: The *direct* method, which uses the indexed data tree to evaluate a query; the *schema* method, which uses the path-tree (schema) of the data tree to find the best k transformed queries (second-level queries); and the *hybrid* method, which is similar to the *schema* approach, but merges second-level queries with similar substructures. However, the approXQL query engine is more than an implementation of the three query-evaluation methods. It consists of all components necessary to load, store, index, and display XML documents. Moreover, it supports the generation of concise abstracts based on the logical documents selected by a query. The system also provides a graphical query editor.

Figure 9.1 on the following page shows the architecture of the system and the data flow between the modules. On the server side, there are six main modules: The database kernel, the document loader on top of the XML parser, the indexer, the query parser, the query processor, and the abstract generator. On the client side there is a graphical query editor controlled by a World Wide Web (WWW) browser. We describe the server-side modules in the following section, and introduce the query editor in Section 9.2.

9.1 The Server-Side Modules

The server-side modules of the approXQL system belong to two separate application programs: The `db-loader` is a command-line application that reads XML documents from the file system or via the HyperText Transfer Protocol (HTTP) protocol, and stores them in the database. It

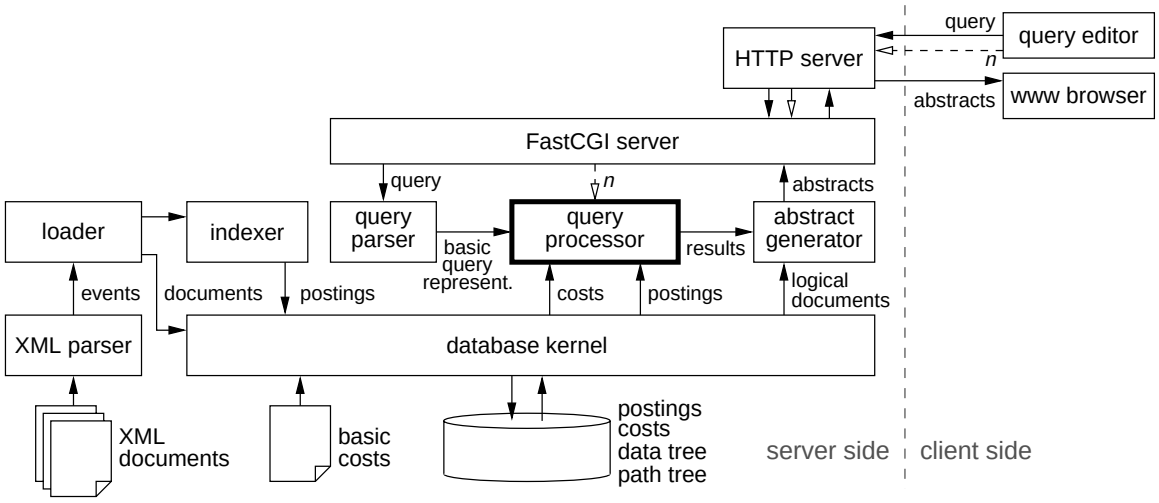

Figure 9.1: Architecture of the approXQL query engine. The arrows show the data flow between the modules; dashed lines with empty arrowheads indicate the flow of control data. The letter n denotes the number of requested results.

consists of the modules XML parser, loader, and indexer. The `db-query` application runs as a background process and communicates with the FastCGI server. It consists of the modules query parser, query processor, and abstract generator. Both application programs, as well as several auxiliary tools, share a common database kernel. All server-side modules are implemented in C++. We first describe the kernel module, then continue with the modules of the two application programs.

9.1.1 Database Kernel

The database kernel is implemented on top of the Berkeley Database toolkit [BER02]. The toolkit provides fundamental database-access methods including sequential files with fixed and variable-length records, queues, B-trees, and extended linear hashing [Lit80]. It also provides subsystems used in building a transactional data store: a logging subsystem, a locking subsystem, and a transaction subsystem on top of both of them. A memory pool subsystem allows shared and cached access to the database files.

All indexes of the approXQL query engine are implemented as B-trees. The leaves of the B-trees store the postings, which are transfered into a memory area managed by the caching subsystem of the Berkeley toolkit. The LRU (least recently used) strategy provided by the cache ensures that only frequently used postings are in the cache; all other postings remain on disc. By analyzing the logs and by varying the cache size, the Berkeley toolkit can be adapted so that the ratio of cache misses is minimal, given a fixed memory size.

To display results and to generate abstracts (see Section 9.1.5), the kernel also manages the data tree built from the XML documents passed by the loader. The documents are passed as sequences of nodes, where each node is annotated with its preorder number and its bound (the preorder number of the rightmost leaf of the subtree rooted at this node). The preorder-bound pair indicates the position of the node in the data tree. Given the preorder number of a result node, the logical document rooted at this node can be retrieved quickly using a B-tree. The last node in the document is indicated by the bound of the root.

In addition to the indexes and the data tree, the kernel also manages the path tree of the data tree. The nodes in the path tree are annotated with their insertion costs and path costs (see Section 8.1). For performance reasons, the path tree is read completely into memory during the initialization of the kernel. At this time, the kernel also builds the in-memory indexes to quickly access all path-tree nodes with a certain type and value.

The interface of the database kernel consists of four parts: the loader interface, the query-processor interface, the costs interface, and the document interface. The loader interface is event oriented. It provides methods to manage transactions and to successively write the constituents of serialized XML documents into the database. The query-processor interface provides access to the primary index of the path tree and to the primary and secondary indexes of the data tree. The costs interface allows to define and access the basic costs. Currently, all cost definitions are specified in a file; the interactive definition of costs is not supported. The document interface allows to access logical documents with given preorder-bound pairs.

9.1.2 Loader and Indexer

The loader and indexer are on top of the kernel module. The loader uses the Xerces XML parser [XER02] to import XML files. It implements the SAX [Meg98] interface to the parser. The SAX interface provides callback methods that are triggered by the parser if certain XML content types (like elements or character data) have been read.

The loader splits up sequences of character data into tokens, assigns data types to them, and stems tokens of type *text* using the Porter algorithm [Por80]. It immediately passes all words, numbers, attribute values, and attribute names to the indexer. Whenever the loader gets an event indicating a start-tag of an element, it passes the name of the element to the indexer. The indexer first identifies the posting belonging to the name. Next, it adds both the preorder number of the element and its distance from the root of the data tree to the end of the posting. At this point, the bound of the element is not yet known. Therefore, the

loader pushes a pointer to the posting entry on a stack. If the end-tag of the element has been announced, then the loader passes the bound and the pointer to the indexer; which adds the bound to the entry the pointer refers to. Note that — despite of the use of an additional stack — the time to construct the index is linear in the number of elements, attributes, and tokens in a document.

The loader also updates the data tree by sequentially writing the nodes created for a parsed document into the database. The preorder numbers of the nodes are inserted into a B-tree to allow the reconstruction of logical documents, as described in the previous subsection.

To speed up the loading-process, the indexer manages a write cache with adjustable capacity. The cache holds all recently extended postings, and thus avoids the slow one-by-one writing of single posting entries. Whenever the cache space is exhausted, the postings are written to the database as large blocks.

9.1.3 Query Parser

The query parser maps an approXQL query to the basic form of its expanded representation (see Section 6.2.1). If restrictions or relaxation are defined for the query, then the module annotates the nodes in the expanded representation with sets of transformation modifiers (see Section 4.4).

9.1.4 Query Processor

The query processor is the main module of the approXQL system. It implements the *direct*, the *schema*, and the *hybrid* query-evaluation method. Figure 9.2 on the next page shows the submodules of the query processor and the data flow between the submodules. Dashed lines with empty arrowheads indicate the flow of control data.

The *expander* submodule derives the expanded query representation from the basic form supplied by the query parser. To encode the permitted deletions, permutations, and value changes, it accesses the *basic-costs index* of the database kernel via the costs interface. Given the type and value of an *s*-node, the index returns the deletion cost and the set of value-cost pairs representing all permitted value changes. Given a pair of values of type *struct*, it returns the cost of permuting nodes carrying these values.

The expanded query representation is passed to the *plan generator*, which creates a query-execution plan as described in Section 6.4. Depending on the evaluation mode (direct or

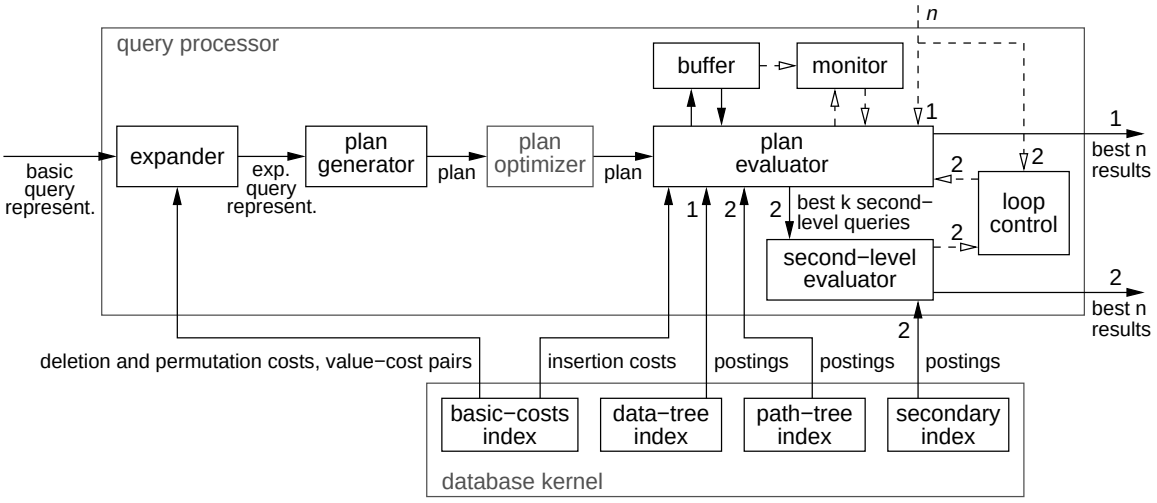

Figure 9.2: The query-processor module and the indexes of the database kernel. The arrows show the data flow between the submodules; dashed lines with empty arrowheads indicate the flow of control data. The numbers assigned to the arrows refer to the query-evaluation methods (1 for *direct* evaluation, 2 for *schema* or *hybrid* evaluation). Arrows without numbers show data flows common to all methods.

schema-driven), the generated plan consists either of basic list operators or of k-operators for extended lists (see Section 7.2.3).

The subsequent *plan optimizer* tries to rearrange the plan operators in order to lower the evaluation times. Note that this submodule does not implement a fully fledged cost-based optimizer. Rather, it uses a simple heuristics to minimize the number of operators using the equivalences of Lemma 6.1 on page 81.

The *plan evaluator* evaluates the query-execution plan. Depending on the evaluation mode, it passes either the indexes of the data tree (*direct* evaluation, indicated by number 1) or of the path tree (*schema* or *hybrid* evaluation, number 2) to the selection operators within the plan. In both cases, the insertion and path costs for data-tree or path-tree nodes are fetched from the basic-costs index. The result of the evaluation is either a sorted list of n node-cost pairs, where n is the number of requested results, or a sorted list of k second-level queries.

If one of the schema-driven evaluation methods is used, then the second-level queries created by the plan evaluator are passed to the *second-level evaluator*, which accesses the *secondary index* of the database kernel in order to find the results for those queries. The sorting order of the second-level queries produced by the *hybrid* query-evaluation method does not necessarily reflect the sort order of the results selected by the queries. Therefore, the submodule also implements an algorithm that inserts new results into the result list and determines the prefix of "save" results, which are then passed to the abstract generator.

The adaptive *buffer* stores intermediate results in order to implement dynamic programming (see Section 6.6.2). For the *schema* and the *hybrid* query-evaluation method, this submodule also stores the results of partially evaluated second-level queries, in order to use them for the evaluation of subsequent second-level queries with identical parts.

The *loop control* submodule controls the submodules for the creation and evaluation of second-level queries. A user-definable function estimates the parameter k based on the parameter n. The submodule triggers the creation of second-level queries. If the first k queries did not retrieve n results, then k is increased using another user-definable function, and the next iteration cycle starts. The incremental evaluation ends if either all results are found, or if subsequent iteration steps do not increase the number of second-level queries.

The plan evaluator is controlled by a resource *monitor*. It ensures that both the maximum evaluation time and the buffer size are not exceeded. The monitor also triggers the termination of the evaluation process if the maximum number of second-level queries are created, or if the second-level queries consume too much memory.

9.1.5 Abstract Generator

Recall that the query processor produces result lists in which each entry represents the root of a query embedding. The query processor additionally tracks the image of the cheapest transformed query tree for each entry. The images are used by the abstract generator. For each entry in the result list, this module selects the logical document rooted at the data-tree node represented by the entry, and produces an abstract of it. The abstract consists of all nodes that are part of the embedding image, together with a match context for each node. The match context of a keyword consists of the m preceding and m following keywords, where m is a number specified by the database administrator. The match context of an attribute name consists of the attribute value, and of the name of the element to which the attribute belongs. For each matching element name, some or all attributes are included in the match context. If a query leaf maps to an element name, then a part of the character data included in the element is added to the match context. All abstracts belonging to entries in a predefined list segment are created and displayed to the users.

The abstract generator is not restricted to a predefined output format. Instead, it provides an interface that can be implemented to produce abstracts in arbitrary formats. Currently, the system supports the output of plain text, HTML documents, and XML documents.

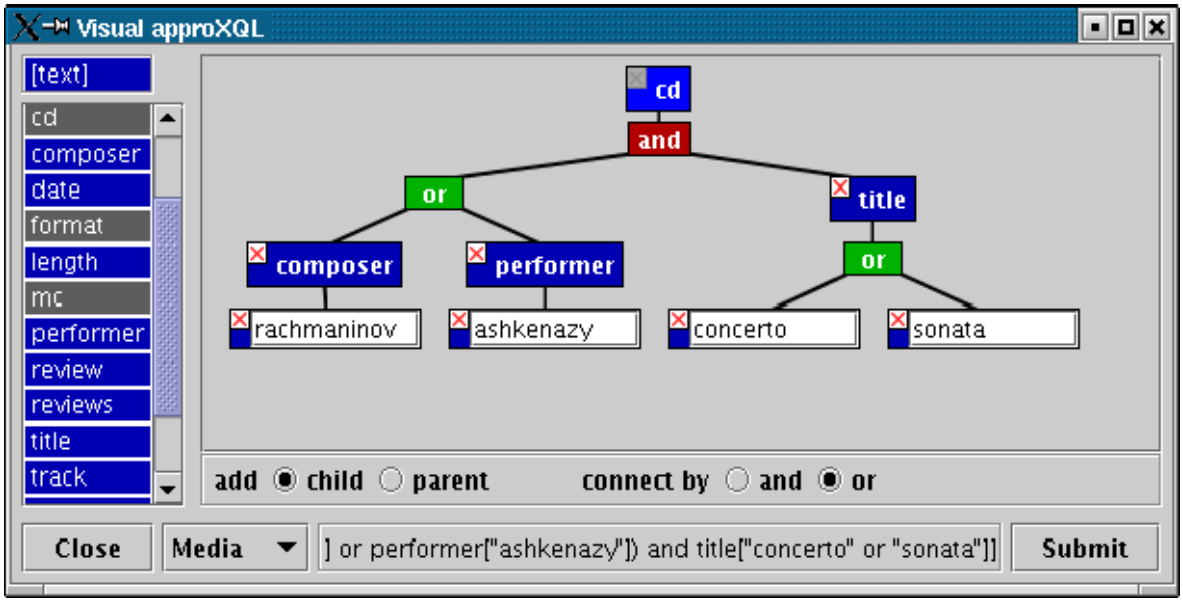

Figure 9.3: The graphical query editor.

9.2 The Graphical Query Editor

Our prototype provides a graphical query editor implemented as a Java applet (see Figure 9.3). The editor displays a list of all element and attribute names that may be used in a query. In the simplest case, the list consists of all different element and attribute names found in the indexed documents. The database administrator can modify this list in order to ease query formulation: First, they can remove all names from the list which are not useful for typical user queries. For instance, they may drop list and table constructors, or elements that emphasize text sequences. Second, the administrator can replace element or attribute names by more descriptive labels. For instance, they may replace the label cd by compact disc. The query processor maps the synonyms to the original names using *zero-cost value changes*.

To construct a query, the user selects a name from the list. The editor displays the selected name as a labeled node in the tree panel. Depending on the construction mode ("add parent" or "add child"), the list of names is adapted so that all names not reachable from the selected name are disabled. The reachability information is provided by the path tree, which is requested from the database kernel. Fields for data input can be added to the query whenever the currently active query selector matches a path-tree node that has a descendant of type *data*. The constructed query — which is additionally displayed in approXQL syntax — is then submitted to the query engine.

The query editor can handle the path trees of several XML document collections. The user

can freely switch between the collections and formulate a query for each. When the user selects a collection a second time, the editor restores the graphical representation built for that collection.

Currently, the graphical query editor only supports the basic syntax of approXQL introduced in Section 3.1. We plan to extend the editor so that the user can specify restrictions and relaxations of query transformations (see Section 3.2). Furthermore, in addition to the alphabetical listing of element and attribute names, we plan to add a second list that displays only those names that have been used most frequently during the previous sessions. The selection frequency of a name may be counted either globally or separately for each user.