The decomate result optimizer

Hoppenbrouwers, J.J.A.C.

Published in:
Proceedings of the Final Decomate II Conference

Publication date:
2000

The Decomate Result Optimizer

Jeroen Hoppenbrouwers
Infolab, Tilburg University
hoppie@kub.nl

Abstract

The Result Optimizer is a relatively independent module in Decomate II that has the task of optimizing the results returned by the main database back end. Optimizing includes result set merging, de-duplication, ranking, and caching. This article explains the design of the Result Optimizer and summarizes most of the algorithms used.

1 Result Optimizing in Decomate II

Within the Decomate II Project[1], a large number of bibliographical and full-text databases have been unified under a generic query protocol and user interface (Place, 1999). The raw results that are returned by the various database back ends through the Multi-Protocol Server (MPS) can be optimized in several ways (Hoppenbrouwers and Paijmans, 2000). Decomate II concentrates on four optimizations:

Result Set Merging If a query is sent to multiple databases, each database will return its own result set. The Result Optimizer must merge these separate sets into one virtual result set, while maintaining the individual references to the databases for future use.

De-duplication When queries are sent to multiple databases, and in practice also within one database, there is a high probability of receiving multiple references to the same document. If we want to represent the multiple databases as one single virtual database, de-duplication of the results is required.

Ranking The merged and de-duplicated result set should be ranked in one of several possible orders. Likely ranking candidates are on plain data fields, such as AUTHOR and YEAR, and more advanced ways such as by relevance to the query. This advanced ranking needs more information than what is available in the result set(s); this information must be supplied by the Broker, possibly through the Concept Browser.

Document Tracing When users query a selection of databases (possibly one single database), it can happen that a document reference is retrieved for which no full text is available in the selected database(s). However, full text might be available via pointers to full text in another Decomate II database. The Document Tracer should alert users to this situation and add links to the full text to the result.

[1] http://www.bib.uab.es/decomate2

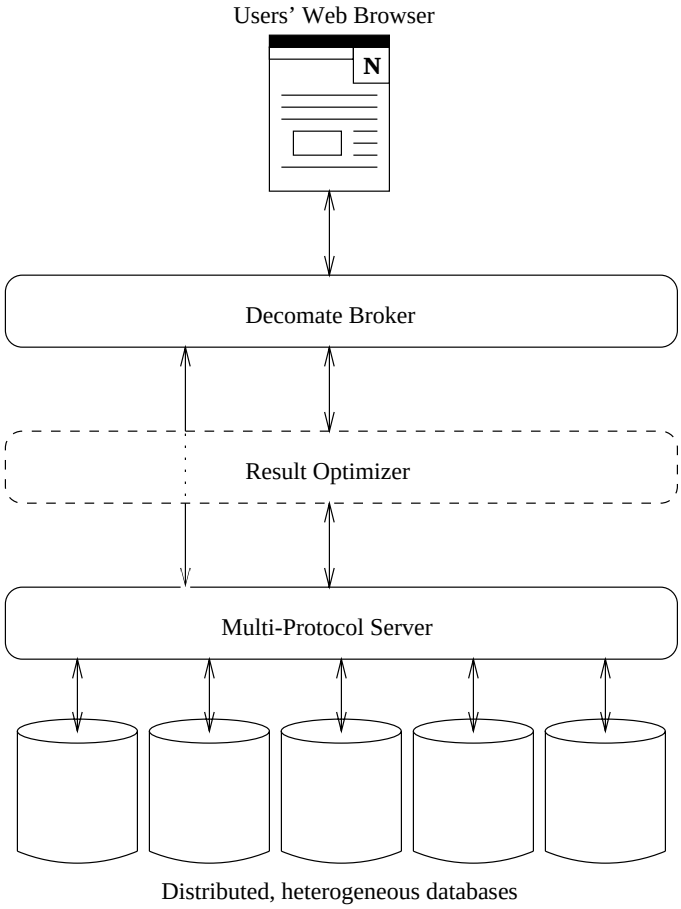

Figure 1: Decomate II Architecture

In this paper, the first two optimizing options will be discussed in detail. Ranking will be touched upon, and Document Tracing will not be presented at all. We refer to other Decomate documents and future developments for this information.

The Decomate II architecture can be summarized as shown in Figure 1. The user has a Web browser that accesses the Decomate Broker. The Broker queries the server back ends through the Multi Protocol Server, formats the results in HTML, and returns them page by page to the user (Hoppenbrouwers and Paijmans, 2000).

Key feature of this architecture is that result sets are not handled in one big transaction, but partitioned in pages, alike the Z39.50 protocol. The length of a page can be tuned by the Decomate II maintainer and typically is 15 references. After a query, only the number of hits in each database is returned and passed back to the Broker. It depends on the configuration of the Broker what happens next. Usually, the user is presented a list, enumerating the hit counts per database so that (s)he can select which database results to view. A subsequent request to the Broker causes a page of results to be retrieved from the database, usually in brief record format. Subsequent drill-down clicks then retrieve full records one by one.

Note that neither the Broker nor the MPS store the result list of any database query—they function on a pass-through basis and only maintain limited session memory to implement 'go back'-functionality. This complete lack of result memory significantly affects the capability of the Decomate II system to execute operations on complete result sets. Therefore we had to add another module to the Decomate architecture which major purpose in life is to add this result set memory and carry out operations on it.

As shown in Figure 1, a new module is placed in between the MPS and the Broker. This is the only place where the complete result list of a query can be captured. It also means that the new module, even if it does not do any actual result optimization, can function as a *proxy cache* for the MPS. It relieves the database back ends from repetitive query execution[2] when a user requests another page of an already executed query. This is an unintended side-effect of the Optimizer, but it can be quite useful at times. More complex implementations could add background query execution, where queries from (especially remote) databases are still running and are still being collected by the Optimizer, while the front-end can already retrieve the first 'page' of results (not yet truly optimized though).

This architecture also provides for some graceful 'fail active' degradation in case of system failures. When the Optimizer breaks down, the Broker can simply bypass the Optimizer and contact the MPS directly. This causes the system to drop all result optimizing functionality, but the queries will still get to the MPS and the user interface will not be influenced in a way that makes work impossible.

Lastly, a separate implementation of the Result Optimizer offers better scalability. Result optimizing is a memory and CPU intensive process. The split design allows to run the Optimizer on a separate machine, where it does not degrade the performance of the MPS. Whereas the MPS takes care of the typical network problems, the Optimizer can concentrate on its core task and since it stays within the local area network, it will experience far less communication problems.

2 The Optimizing Process

Two aspects mostly influence the architecture and design of the Optimizer: action sequencing and response time requirements. Action sequencing means that some actions cannot be performed before other actions have been fully completed. The most obvious example of this is the actual de-duplication; this cannot take place on partial result sets. Response time requirements dictate that no default action should ever take more than a few seconds, preferably less than two. This excludes activities such as the tracing of all full text documents for all returned references.

Another design factor is the browser-oriented architecture of Decomate II. It is impossible to send lagging updates to a browser; any updates must be triggered by a user action. For example, it is not possible to present a page of references and fill in full text links 'on the fly' while the user is reading the page.

A complete optimizer run consists of several stages that are sequenced. The Broker first sends a search request to the MPS. The Result Optimizer intercepts this request, strips out the Optimizer Data Block, and forwards the rest of the request to the MPS (although it may store the query for future use). The MPS performs the query and sends back the results, typically a list of hits per queried database.

[2] Or repetitive partial result set submission, if they support both search and present requests.

The Result Optimizer takes in this hit list and issues a present request to the MPS, ordering up all full records previously retrieved by the databases. The Result Optimizer then caches the complete result sets and fires the merger, the de-duplicator, and the ranker. After processing, the Optimizer forwards the number of 'virtual hits' to the Broker and leaves the virtual result set in the cache.

Depending on user actions, the Broker comes back to the Optimizer with requests for presentation of specific pages, one at a time. Users might want to access the original database sets, in which case the Optimizer passes the request straight through to the MPS. Requests for pages from the virtual result set are served from the cache, and therefore very fast. For each page, the Optimizer might need to run the Document Tracer, after which the enriched result can be passed to the Broker.

The result cache expires after a given timeout, or when the Broker signals that the session is being canceled. Note that a flushed cache can be completely restored by re-issuing the MPS query, the Optimizer run, and the Document Tracer run.

2.1 Virtual Result Sets

A common feature of all activities by the Result Optimizer is that they produce a new result set, which is deduced from possibly several 'original' database result sets. We call this new set the *virtual result set*, because it is artificially generated and only exists in the Result Optimizer, not in any contributing database.

However, from the Broker's point of view, the optimized virtual result set is a normal result set like any other. This implies that the Broker should *ask for the virtual result set* just as it asks for any other set. The Broker should provide a proper result set ID and enough information to construct the virtual result set, including de-duplication limits and ranking information.

A standard database chunk of XML, such as included in all search requests, looks like the following.

```
<database>
  <DatabaseName>olc</DatabaseName>
  <DatabaseServer>tcp:dbiref.kub.nl:1289</DatabaseServer>
  <ResultsetNaming>Yes</ResultsetNaming>
  <Profile>usmarc</Profile>
  <Authentication>***********</Authentication>
</database>
```

For the virtual result set, this database block is replaced by a specialized 'Optimizer' block which contains the required information for the Result Optimizer:

```
<Optimizer>
  <OptimizeLimit>100</OptimizeLimit>
  <RankingInfo>
    <Order>Relevance</Order>
    <Keyword>
      <Term>Tinbergen</Term>
      <Weight>0</Weight>
    </Keyword>
    <Keyword>
      <Term>Value chain</Term>
      <Weight>1</Weight>
    </Keyword>
    <Keyword>
      <Term>Micro-economics</Term>
      <Weight>-1</Weight>
```

```
    </Keyword>
  </RankingInfo>
</Optimizer>
```

Usually, the database chunks are encapsulated in a search or present request, and preceded by the standard XREP fields such as session ID and result set ID.

2.2 The Optimize Limit

By tuning the OptimizeLimit, the Broker can control how much effort and time the Result Optimizer will spend on creating the virtual result set. During a typical first search, the Broker sets the limit to a low number of hits, such as 20. This means that the Result Optimizer will only create the virtual result set when the total number of hits of all individual result sets together is 20 or less. Since optimizing requires a download of all full records from all result sets, followed by relatively costly comparison and ranking operations, optimizing should not be done by default on result sets that contain more than a limited number of records. Whenever the Optimizer decides not to optimize, the number of hits in the virtual result set is reported back to the Broker as zero.

A subsequent search request from the Broker (usually initiated because the user decided that the retrieved result sets contain a number of records that is sufficiently low to wait for optimizing) can have an OptimizeLimit that is high enough to equal or exceed the total number of hits in all individual result sets. This forces an optimize run. Local considerations and available processing power for the machine that runs the Optimizer might require other limits for optimizing, such as 100 for the lowest and 1000 for the highest allowed values. Users might be offered to set the limit themselves as well, guarded by a system-wide upper limit.

The advantage of this preset OptimizeLimit is that the clients themselves (users through the Broker) can decide what waiting time is acceptable. Any other method, such as a fixed limit of 150, will take this decision away from the users, and makes them wait unnecessarily long when doing 'probe queries' while preventing lengthy but requested optimizing runs on large result sets.

2.3 Handling Virtual Result Sets

Three features set virtual result sets apart from the standard database sets: they are computationally expensive to create (download time plus modification time), they are memory-intensive because they need to be stored locally in the Result Optimizer, and they potentially obscure information from the user after de-duplication.

After creation, a virtual result set must stay available for a period of time in which the Broker can issue present requests for parts of the set. To save memory, result sets are only stored for a specific amount of time, using the standard expiration approach. On top, when the Broker gets notified specifically that a user quits and sends a 'delete session' message, all result sets belonging to this session are destroyed. The cache part of the Result Optimizer keeps tabs on which result sets are locally cached and which requests must be passed on to the MPS. Unknown requests are always passed on.

Currently, the Result Optimizer *does never delete information*. The de-duplication process only groups equal records together, treating them as one big record with subdivisions. All information that comes out of the databases is retained and passed on to the Broker. It is up to the Broker to decide which part(s) of the record to display. Most current implementations retain the same basic grouping idea, and present equal records

to users as large records with more than one full document record in them. The users then can visually determine whether the grouped records are actually duplicates or not, and even select which of the databases to go to for the full document.

3 De-duplicating Result Sets

Previous work has concentrated mainly on de-duplication to detect *exact* duplicates, either for final bibliographical database merging or for copy detection (Campbell et al., 2000). Much effort is spent on finding (near) perfect hash functions to condense documents or records into small, but (near) unique signatures. However, in the Decomate II system, the de-duplication process must have a considerable degree of fuzziness to allow for *close* duplicates. Various databases (organizations) have slightly different indexing procedures. Some might store full author first names, while others only store their initials. The de-duplicator must catch these slight variations.

3.1 Starting Point

Since 100% accurate de-duplication is a difficult, if not impossible task, it seems acceptable to aim for a performance that approaches 80%. We assume that having 20% of all duplicates still in the result set is less bad than having 20% of non-duplicates wrongly grouped with another record, so the main aim of the algorithm should be to *flag duplicates if and only if there is a high degree of overlap between them*.

Analysis of several bibliographical databases available through a Decomate system leads us to select four main 'fields' to classify document records with: authors, title, source, and year of publication. Most databases provide these fields right away; in case of mismatches, mapping instructions can be retrieved from the meta-database. More specific fields, such as page ranges and publisher, can either be added to the source or be skipped right away (the 80% rule).

3.2 Field Smashing

All incoming fields are first *smashed*. This is a mostly generic procedure that does the following to the incoming string:

- Convert it to lower case.
- Remove HTML escaped characters.
- Remove all non-alpha-numerical characters.
- Suppress all runs of one to three characters.
- Collapse all runs of more than one space into one space.

Provisions for exceptional processing according to field type have been made, but currently only the *Source* field (usually containing the journal name) is treated differently. This is because journal names often contain subtitles. The *Source* smasher only smashes the journal name up to the first occurrence of a colon ':'. Everything after the colon, including the colon itself, is disregarded. Improvements can be expected when characters with diacriticals are not removed, but replaced by standard characters (ë by e, for example).

After individual field smashing, the resulting smashed strings are concatenated and sorted alphabetically (removing doubled elements), so that *signature strings* remain.

As an example, several existing bibliographic references are smashed into signatures in the next section.

3.3 Examples

Original Reference Boll, S.; Klas, W.; Battaglin, B. Design and Implementation of RMP - A Virtual Electronic Market Place. SIGMOD record : a quarterly publication of the ACM Special Interest Group on Management of Data (T06853) vol.27 (1998) nr.4 p.48-53

Authors boll klas battaglin

Title design implementation virtual electronic market place

Source sigmod record

Year 1998

Signature 1998 battaglin boll design electronic implementation klas market place record sigmod virtual

---

Original Reference Sussman, H.M.; Dalston, E.; Gumbert, S. Original Papers - The Effect of Speaking Style on a Locus Equation Characterization of Stop Place of Articulation. Phonetica : internationale Zeitschrift fuer Phonetik (T05374) vol.55 (1998) nr.4 p.204-225

Authors sussman dalston gumbert

Title original papers effect speaking style locus equation characterization stop place articulation

Source phonetica

Year 1998

Signature 1998 articulation characterization dalston effect equation gumbert locus original papers phonetica place speaking stop style sussman

---

Original Reference Flora, Jan L. Presidential Address - Social Capital and Communities of Place. Rural sociology : devoted to scientific study of rural and small-town life (T00202) vol.63 (1998) nr.4 p.481-506

Authors flora

Title presidential address social capital communities place

Source rural sociology

Year 1998

Signature 1998 address capital communities flora place presidential rural social sociology

---

Original Reference Andre, E.; Rist, T.; Muller, J. WebPersona : a lifelike presentation agent for the World-Wide Web. Knowledge-based systems (T08576) vol.11 (1998) nr.1 (September) p.25-36 (24 refs.)

Authors andre rist muller

Title webpersona lifelike presentation agent worldwide

Source knowledgebased systems

Year 1998

Signature 1998 agent andre knowledgebased lifelike muller presentation rist systems webpersona worldwide

3.4 Matching signatures

Smashing already provides a high amount of protection against differences in record entry, but it cannot protect against typos in names or titles. Therefore, the current system implements the following algorithm, which provides for some margin of error in record matches. It is also designed for efficient execution.

- Every element of a signature is added to a lookup table, which is an index from element string to record ID with multiple IDs per element possible. Essentially, the lookup table is the inverted index of all signature elements.
- Using this inverted index, a second table is built that contains the number of matches between each individual element of a signature and all other signatures that contain the same element. If two signatures share five elements, their match counter reads five.
- Additional tables are created to form groups of records that potentially match (using transitivity, if 34 matches 45 and 50, 45 matches 50 as well). 'Potentially' means here a rough comparison of signature lenghts; they must be within 20% equal to be included in the group. This is done to avoid comparing signatures with huge size differences, which is pointless.
- A limit value is calculated by multiplying the number of elements of the signature by (currently) 0.85 and rounding to the nearest integer. This is done for each potentially matching pair of signatures, taking the shortest of them as reference (but they will be within 20% equal in length anyway). Signatures below a certain length, currently 5, can never lead to a match.
- All signatures that share more than their limit value of elements with another signature (quickly determinable through the match table) are considered equal, and therefore their original index records are considered to represent double documents. They are gathered in a group. Group merging occurs as soon as required.

The first three steps (together with the smashing process) can be taken on the fly, as the full records of the databases stream in over the network. CPU capacity usually is sufficient to outrun the network. The last two steps need to be executed when all records have been received and indexed.

4 Ranking Result Sets

Ranking of results can be done both by the individual databases and by the Result Optimizer. Although most databases offer the common forms of field ranking (on author, title, etc.), this feature is of limited use since the individual result sets must be merged in the end. Merging sorted lists is somewhat easier than merging unsorted lists, but it is not at all sure that the individual lists are sorted in *exactly* the same way (díãçrïtìçåls come to mind). The bottom line is that for merged result set ranking, the Result Optimizer needs to implement a completely self-sufficient ranking module.

The current Optimizer does not include ranking of result sets that remain on the contributing servers; only the cached, merged, de-duplicated virtual result set is ranked by the Result Optimizer. This means two things:

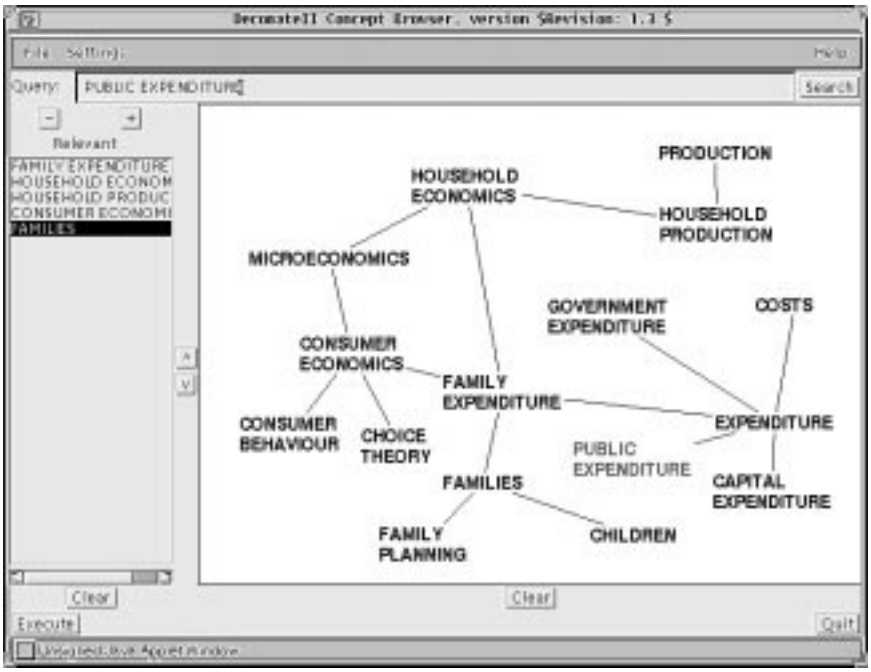

Figure 2: The Concept Browser

1. Present requests for individual result sets will only feature rankings if they are implemented by the remote databases.

2. Ranking of the virtual result set can only be done after the whole set has been compiled.

The latter is not a surprise, and given the fact that virtual result sets are compiled on explicit user request only (or by default if their size is less than a low limit), we think that most users gladly accept a waiting time of several seconds to fully optimize a long list of document records. Preliminary user studies indicate that this assumption probably is right.

Two basic types of result ranking can be distinguished: string order and relevance order. String order is a more or less complex form of the standard alphabetical-lexicographical ranking algorithm that has been implemented thousands of times already. It will be used mainly for the author and title fields, and possibly for a reversed 'order of acquisition or publication' listing (latest/newest on top).

Relevance order is much more complex, because it does not only involve simple string ranking but also concept matching and weighing. Since it is difficult to perform proper relevance ranking with the limited information available in standard keyword queries such as `Tinbergen and micro-economics` (Hoppenbrouwers and Paijmans, 2000), we need more extensive data to begin relevance ranking in the first place.

The *Concept Browser* (Fig. 2), an alternative user interface that is intended to guide users through standard thesauri in order to get better queries, can provide this extra ranking information. However, this information must be conveyed to the Result Optimizer through the Broker in some way. We use the Optimizer Block to hold this

information. A preliminary variant of the Optimizer Block including ranking information can be found in Section 2.1.

Through the Concept Browser, users can select thesaurus keywords using a two-dimensional interface and collect query terms in a 'term bucket'. Optionally, this bucket can be given a ranking by moving terms up or down in the list. Users who decide not to provide weight information therefore will not receive relevance ranked result lists. We might experiment with default rankings according to the position of words in the Conceptual Network, or using statistical techniques from the field of Information Retrieval (Hoppenbrouwers and Paijmans, 2000). But ranking the term bucket manually is a relatively simple and clear mechanism, which we hope will be understandable by the users and at least helps them creating queries with the *potential* for better precision at the top.

5 Conclusions and Future Work

The Result Optimizing functionality required in the Decomate II system has been implemented as a stand-alone multithreaded server process that can be distributed to any machine on the network. This approach turned out to be flexible and scalable enough for production requirements.

The used de-duplication algorithm seems to work well for most databases, but is still not sufficiently thorough for all combinations of all available databases. Further tuning is required to take care of specific formats and fields. We expect that eventually the de-duplicator will score significantly above the 80% correctness that we set as a target at the beginning of the project.

Future work will include further development of the Concept Browser and the link with the relevance ranking, in-depth research into the qualitative and quantitative performance of the de-duplicator, and extensive user feedback to investigate whether the offered services meet the users' demands and expectations.

References

Campbell, D. M., Chen, W. R., and Smith, R. D. (2000). Copy Detection Systems for Digital Documents. In Hoppenbrouwers, J., de Souza Lima, T., Papazoglou, M., and Sheth, A., editors, *Proceedings of the IEEE Advances in Digital Libraries 2000 (ADL2000)*, pages 78–88. IEEE Computer Society.

Hoppenbrouwers, J. and Paijmans, H. (2000). Invading the Fortress: How to Besiege Reinforced Information Bunkers. In Hoppenbrouwers, J., de Souza Lima, T., Papazoglou, M., and Sheth, A., editors, *Proceedings of the IEEE Advances in Digital Libraries 2000 (ADL2000)*, pages 27–35. IEEE Computer Society.

Place, T. (1999). Developing a European Digital Library for Economics: the DECOMATE II project. *Serials*, 12(2):119–124.