

# Ad-hoc Queries over Document Collections - a Case Study (incomplete workshop discussion draft)

Alexander Löser<sup>1</sup>, Steffen Lutter<sup>1</sup>, Patrick Düssel<sup>2</sup>, Volker Markl<sup>1,2</sup>

<sup>1</sup> DIMA Group, Technische Universität Berlin, Einsteinufer 17,  
10587 Berlin, Germany

(firstname.lastname@tu-berlin.de)

<sup>2</sup> Intelligent Data Analysis, Fraunhofer Institute FIRST,  
12489 Berlin, Germany

(firstname.lastname@first.fhg.de)

**Abstract.** We discuss the novel problem of supporting analytical business intelligence queries over web-based textual content, e.g., BI-style reports based on 100.000's of documents from an ad-hoc web search result. Neither conventional search engines nor conventional Business Intelligence and ETL tools address this problem, which lies at the intersection of their capabilities. “*Google Squared*” or our system *GOOLAP.info*, are examples of these kinds of systems. They execute information extraction methods over one or several document collections at query time and integrate extracted records into a common view or tabular structure. Frequent extraction and object resolution failures cause incomplete records which could not be joined into a record answering the query. Our focus is the identification of join-reordering heuristics maximizing the size of complete records answering a structured query. With respect to given costs for document extraction we propose two novel join-operations: The multi-way CJ-operator joins records from multiple relationships extracted from a single document. The two-way join-operator DJ ensures data density by removing incomplete records from results. In a preliminary case study we observe that our join-reordering heuristics positively impact result size, record density and lower execution costs.

**Keywords:** information extraction, document collections, query optimization

## 1 Introduction

We address the problem of executing structured queries over extracted records which are obtained from document collections defined at query time, usually through keyword search queries on the document collection. Let us illustrate that by an example of our system *GOOLAP.info*. Triggered by a structured query, *GOOLAP.info* generates a set of keyword queries, collects web documents, extracts information, joins the extracted records and presents extracted structured results in a tabular form. Extracted data then can be loaded into a data warehouse and cleansed or further transformed like any other structured data. E.g., for the query “*List persons, their age and position, companies they work in and company technologies*”. Frequently, required information for obtaining a single record answering the entire query is distributed among several documents. As a result, text data from individual

CompanyTechnology		PersonCompanyPosition			PersonAge		Join?
Technology	Company	Company	Position	Person	Person	Age	
IP	ICANN	ICANN	CEO	Paul Twomey	Paul Twomey	47	Yes
Video On Demand	Delta Air Lines	Delta Inc.	CEO	Gerald Grinstein	Gerald Grinstein	71	No
Linux	Oracle	Oracle Inc.	CEO	Larry Ellison	Larry Ellison	51	Yes
cellular telephone	NULL	Vodafone	CEO	Arun Sarin	NULL	53	No

**Table 1: Results after extracting relationships**

documents need to be collected, extracted and integrated. E.g., for this query, records for three relationships are extracted by extractors *CompanyTechnology*, *PersonCompanyPosition* and *PersonAge*. Next, object reconciliation techniques, such as [20], are applied to join columns “*company*” and “*person*” to ensure that values match to the same real world object, even if string representation differs. Finally, records from each relationship are joined to create a result. Table 1 presents extracted records for three relationships.

**Challenges:** Following [7], a result record is complete, if it covers all attributes for answering the query and attribute values are not “null”. However, we observed in our sample application *Goolap.info* that joins could frequently not be executed because extractors return incomplete facts or object reconciliation techniques fail:

- **Extractor returns incomplete facts.** Extractors, such as OpenCalais.org, frequently return incomplete facts to indicate human evaluators the existence of the fact in the document. Typically extractors fail because of *missing textual data* in the document or a *vocabulary mismatch* between the document and the rules of the extractor. E.g., a document could talk about “*Vodafone’s CEO*” but not explicitly mention the person “*Arun Sarin*” (*missing textual data*) or the company technology extractor was not able to detect the company “*Vodaphone*” in the text because of a misspelling or unknown spelling (*vocabulary mismatch*).
- **Object reconciliation failures.** Object reconciliation [20] makes have usage of additional context information given on the same page, e.g., other extracted objects, morphological information, detected anaphora information etc. While object reconciliation often performs well on strings in the same page, it frequently fails when reconciling objects from different pages. E.g., strings “*Delta Air Line*” and “*Delta Inc.*” could not be referenced to the same object.

**Addressed Problems:** Application users have two options to overcome these challenges: They could fine tune their existing extraction and reconciliation techniques, such as proposed in [2, 9, 11, 20]. However, such costly operations involve a significant amount of human labor. We argue that in the rapidly growing web, often documents are available that better match the mechanics of the used extractor and of the object reconciliation technique. Therefore our preferred option is to formulate keyword queries for web search engines with respect to an extractors and

a given query plan that has been computed for a structured query. To define, plan and execute such queries our contribution is three-folded:

- **System Architecture and Operators.** We propose operators for planning and executing structured queries over ad-hoc selected document collections.
- **Join-operations for avoiding incomplete facts.** The two-way join-operator DJ removes incomplete facts during join-execution. The multi-way CJ-operator joins extracted facts only if they have been identified from a single document and refer to the same object.
- **Optimization goal: Result maximization and fact completeness.** Given document collections we investigate heuristics for determining the plan returning a maximum of complete records answering a structured query and evaluate planning heuristics in a case study.

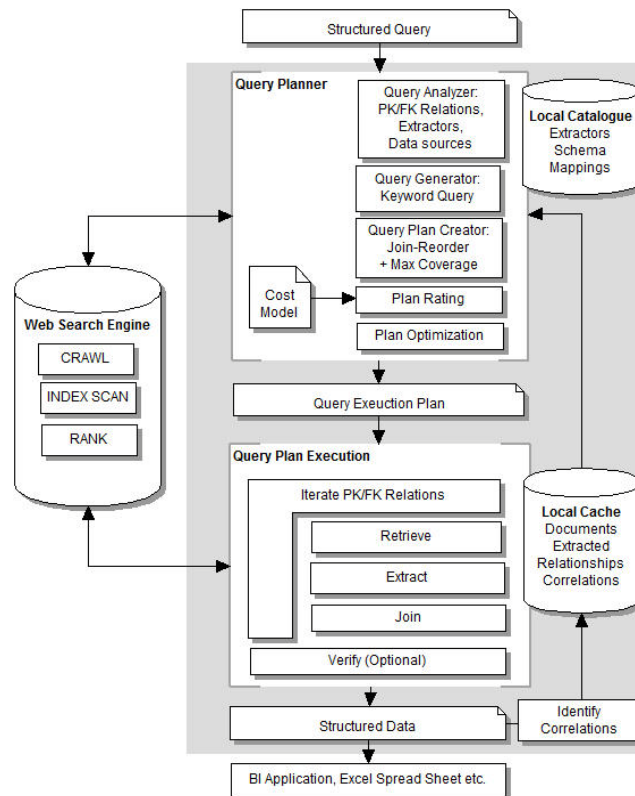
The rest of the paper is organized as follows: Section 2 introduces to query planning and - execution strategies. In Section 3 we investigate our heuristics for maximizing result size on preliminary case study and Section 4 we discuss related work.

## 2 Query Planning and Query Plan Execution

An ideal query planner ensures complete facts records and maximizes result size. This Section describes our system and novel planning techniques to achieve these goals.

### 2.1 System Overview

Executing structured queries over document collections defined at query time requires combining data from two worlds: The first world is the “web world” where unstructured data resides in document collections. Given the huge amount of text data produced on the web, operations, such as crawling, indexing and ranking, require significant infrastructure investments. Therefore setting up an infrastructure will be infeasible for most users. Another option is to leverage existing web search engines and move costs for collecting, storing and updating text data to search engine providers. While their business models prohibit sharing huge amounts of indexed text data, interfaces, such as *Yahoo BOSS*, provide access to small data samples using a keyword query. The second world is the traditional world of structured data. To obtain structured data, similar to an ETL process, information extraction and data cleansing techniques are applied to relevant text documents. Structured data could also result from prior information extraction operations and is stored in a local cache, e.g. a relational data base system. With “enough” and fresh structured data, we could execute structured queries without extracting additional records obtained from document collections at query time. Given these two worlds, the overall performance of our system depends on components shown in Figure 1:



**Figure 1: GOOLAP.info system overview**

**Query planner.** Given a structured query this component determines an execution plan. First, a *query compiler* parses a structured query, identifies extractors, document collections stated in the query and creates an abstract syntax tree. In our *GOOLAP.info*<sup>1</sup> system, each relationship corresponds to exact one extractor which is selected before query planning, e.g., by techniques described in [13]. Further, we assume that all extractors use unique identifiers for identical objects, e.g., used in the system *OPENCALAIS*<sup>2</sup>. Given one or multiple relationships from the structured query, a *query generator* generates keyword queries to obtain relevant documents from web search engines. According to the abstract syntax tree of the query, a *query plan creator* enumerates possible query plans. Finally, according to heuristics and cost models, an *optimizer* chooses a best plan for a given optimization criteria, e.g. a plan maximizing result set size.

**Query plan execution.** According to chosen plan, text data is retrieved, extracted and joined. First, *web search engines* are queried to find top pages relevant for generated keyword queries in the plan. For retrieved text documents *information extraction (IE) engines* identify named entities or relations and resolve extracted objects. Extracted records are often inherently dirty; e.g., they contain duplicate or contradicting records or contain unverified facts. Optional *data verification components* employ data

<sup>1</sup> <http://www.goolap.info>

<sup>2</sup> <http://www.opencalais.org>

cleansing techniques, e.g. methods used in ETL load processes in data warehouses [14]. In addition, extracted relations and obtained documents from prior results are *cached* as decoupled crawl for a particular domain. When a structured query is executed, structured data is obtained from a local cache and is complemented with additional extracted records data collections obtained at query time.

## 2.2 Understanding “Human-powered” Query Execution Strategies

We briefly review human interactions for collecting structured records from text data. Let’s assume a user wants to collect information about companies, technologies and employees of these companies. The following human strategy focuses on collecting such records by iteratively adding tuples in a row-by-row fashion until a certain goal is reached (e.g., time out, enough records are collected etc.):

1. **Select initial text document collection:** One chooses a trusted document collection providing “seed” data. E.g., let’s assume “*en.wikinews.org*” is such a “trusted” source.
2. **Collect relevant text data for a relationship:** Next, one a keyword query to receive relevant documents for a particular relationship, e.g., “*company technology site:en.wikinews.org*”. As a result a ranked document list is retrieved.
3. **“Extract” and “join” structured data from top-ranked pages:** On each text document, one will identify records (*PersonCompanyPosition*, *PersonAge* and *CompanyTechnology*) and will “join” identified relations. To ensure “*relevant*” records, one will only read top-k relevant documents. As result, (potential incomplete) records include values for companies, technologies, persons, positions and their age.
4. **Select next relationship, collect, “extract” and “join” results.** For extracting missing values in records, one will chose a next relationship and text document collection. Let’s assume, one likes to collect *PersonCompanyPosition* records. For each found company in step 3, one will form a keyword query by including the company name and a phrase indicating a *PersonCompanyPosition* relationship. E.g., to collect CEO’s working for “*Microsoft*”, one types “*Microsoft CEO*”.
5. **Iterate over further relationships until done.** One will iterate over next relationships and switch back to step 4, until no further relationships exist. One might terminate this task early, e.g., when enough records have been collected or more than a specific amount of time was spent on this task.

This simple row-by-row strategy employs characteristics minimizing “*human workload*”; e.g., in step 1 and 4, depending on how well keyword queries have been chosen with respect to a relationship, only relevant pages are returned and need to be read. *Record relevancy* is ensured in step 2 by restricting keyword queries o trusted document collections and by examining top result pages only. In step 3 a *coverage-driven joins is executed*, e.g., if a page *covers* records for answering multiple relationships in a query, additional page context (text on page, images etc.) is used to

decide, if identified relations belong to the same object. In step 4 *density-driven joins* seek missing results for incomplete records. To execute queries, certain implicit human knowledge is required, which might not always be available or correct: Join order is based on estimates about the potential “availability” of relevant data in a document collection or based on simplicity to “guess” a keyword query to obtain relevant document collections for a relationship. To model such human processes in a query planner in the rest of this section we identify core operations, nested functions and discuss join-reordering techniques to identify a candidate query plan space.

## 2.3 Elementary Plan Operators

**Overview.** Our system operates on relational data extracted from web pages. To receive structured data we introduce the SEARCH-operator which retrieves a set of documents satisfying a keyword predicate. We denote TOP for an operator that returns the top-n documents from a document collection. The EXT-operator extracts structured relations from a set of text documents (i.e., from unstructured, natural language). Each extracted relation is a table T with k columns. For simulating human query execution strategies on extracted relational tables we reuse relational operators, such as *join* ( $\bowtie$ ), *projection* ( $\pi$ ), *selection* ( $\sigma$ ) and *distinct* ( $\delta$ ). The sub query operator ( $\Sigma$ ) executes an inner query for each record of an outer query where attributes computed by the outer query can be used in the inner query as correlated values (a.k.a., bind-in values). We use the sub query operator together with the KEYWORD function, which generates a set of strings that identify relevant documents for an extractor. Subsequently, we will use the KEYWORD-function to produce search keywords to a SEARCH-operation in order to obtain relevant documents from a large data set (a.k.a. the web) for information extraction. Below we briefly describe interfaces and functionality of the operators and functions that we use in addition to the traditional relational operators:

**KEYWORD(ds, e, kw1)** returns a list of keywords *kw2* to retrieve relevant documents for the document collection *ds* and a set of extractors *e*. If optional keywords *kw1* are supplied to the KEYWORD function as parameter, these keywords *k1* are included as substring in each member of the generated keyword list *kw2*. KEYWORD employs an automatic query-based technique from [12]. KEYWORD is specifically tuned for the behavior of a particular search engine or content management system and its search parameters.

**SEARCH(ds, kw)** retrieves documents from a document collection *ds* that contain a list of keywords *kw*. The keyword parameter *kw* could be a list of constant keywords, or keywords computed by the KEYWORD function as introduced above. Depending on its implementation, this operator may be non-deterministic. The collection of documents and its order may vary when repeating a SEARCH operation with identical keywords.

```

DEF CT(company, technology, docid) AS ('en.wikinews.org', OpenCalais.CompanyTechnology)
DEF PA(person, age, docid) AS ('en.wikinews.org', OpenCalais.PersonAttributes)
DEF PCP(person, company, position, docid) AS ('en.wikinews.org', OpenCalais.PersonProfessional)

SELECT PA.age, PA.person, PCP.position, PCP.company, CT.technology
FROM PA, CT, PCP
WHERE PA.person = PCP.person
AND CT.company = PCP.Company
AND PCP.position like 'CEO'

```

**Figure 2: Example Query  $q1$**

**TOP(ds, k)** returns a document collection  $ds$  (i.e., a relation containing documents) which consists of the top- $k$  documents of the ordered collection  $docs1$ .

**EXT(ds, e)** produces a relation that contains *one or multiple* structured records for each document in the document collection  $ds$  as computed by extractor  $e$ . Basically, EXT extracts structured, semantic information from a text document. For implementations please, e.g., see [2,9,10,11]

**$\Sigma(rOuter, rInner, bi)$**  denotes a sub-query (basically, a nested loop join [18]) operator that executes the  $rInner$  relational sub-query expression for each record of  $rOuter$ . An optional set of bind in attributes  $bi$ , computed in the outer relation can be referenced in the inner relation. In this case, the values of the bind-in attributes of each outer tuple serve as constants in the inner sub-query (effectively making the sub-query correlated [18]).

After introducing our elementary execution operations, we now give a simple example on composing them into a nested operator.

**STE(input, ds, e, A, k)** computes a relation containing extracted information from a document collection  $ds$ . Structured records computed by STE(input, ds,e,A,k) have been extracted by extractor set  $e$ . If an input is supplied, the attribute set  $A$  from the input is used to fine tune the search result for the extraction in combination with an optional tuning parameter  $k$ . The parameter  $k$  is an internal tuning parameter which influences the number of relevant documents [5] that are considered for an extraction. We define  $STE_{complex}(input, ds, e, A, k) =$

$$\Sigma(\pi \text{ KEYWORD}(ds, e, A) \text{ AS } kw \text{ (input) AS } kw, \text{ EXT}(\text{TOP}(\text{SEARCH}(kw, ds), e, k)), kw)$$

However, in many cases STE is used without an input and thus no attribute set  $A$  exists, i.e,  $STE(ds, e, k) = STE(\emptyset, ds, e, \emptyset, k) =$

$$\Sigma(\pi \text{ KEYWORD}(ds, e, \emptyset) \text{ AS } kw \text{ } (\emptyset) \text{ AS } kw, \text{ EXT}(\text{TOP}(\text{SEARCH}(kw, ds), e, k)), kw)$$



**CJ(ds, e1, e2, A, k)** combines the extraction results of two extractors  $e1$  and  $e2$ , if the results have been found in the same document. The join attribute set  $A$  denotes attributes that both extractors share (i.e., identical semantic concepts that both extractors extract independently).  $k$  is again an internal tuning parameter. CJ is commutative. Figure 4 presents an example query utilizing CJ.  $CJ(e1, e2, ds, k, A)$  is defined as:

$$EXT(doc, e1) AS E1 \bowtie_{(E1.docid=E2.docid \text{ and } \forall a \in A: E1.a=E2.a)} EXT(doc, e2) AS E2$$

where

$$doc = STE(ds, \{e1, e2\}, k)$$

**DJ(rOuter, ds, e, A, k)** joins records from the relationship rOuter with records, extracted by extractor  $e$ . Keywords for searching documents from  $ds$  include attribute values  $A$  from rOuter as well as keywords for extractor  $e$ .  $k$  is again an internal tuning parameter. DJ is defined as

$$DJ(input, ds, e, A, k) = STE_{\text{complex}}(\delta A(\pi A(input)), ds, E, A, k)$$

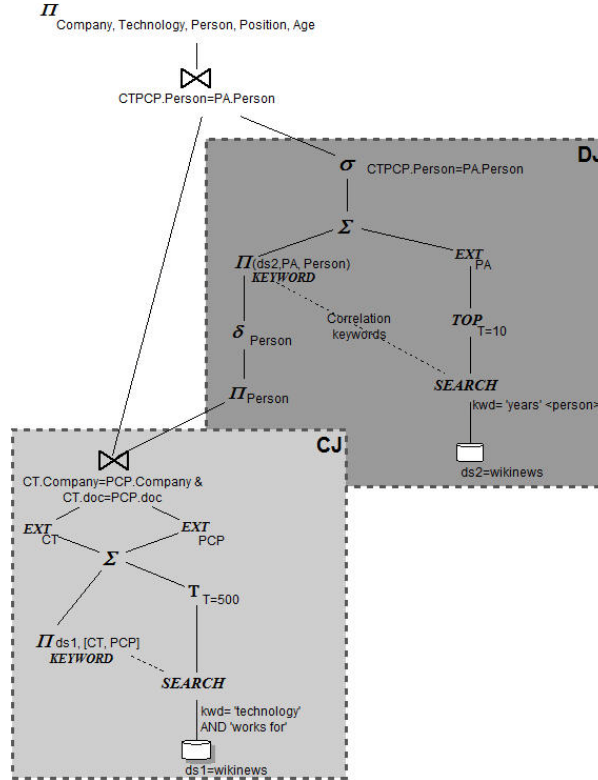
## 2.5 Example query and Example plans

Figure 2 introduces to example query  $q1$ . The query asks to retrieve records of attributes age, person, position, company and technology from three relationships: CT, PA and PCP. Relations are extracted from document collections. Figure 3 introduces an example query plan for executing  $q1$ : First, records for CT are obtained using STE from document collection  $ds1$ . Next, DJ is executed to find matching PCP relations for companies found in CT relations. Results for PCP and CT are joined to a PCPCT relation. Finally, DJ uses persons in PCPCT as input to retrieve PA relations from text collection  $ds3$ . PCPCT and PA relations are joined to PCPCTPA relations. In the plan presented in Figure 4, extractor  $e1$  and  $e2$  share a single document collection  $ds1$  while  $e3$  uses  $ds2$ . First, CJ joins extracted CT and PCP relations identified in the same document via the company attribute. Next, DJ generates PA relations for each distinct company in PCPCT relations. Finally PA and PCPCT relations are joined to PCPCTPA.

## 2.6 Plan Enumeration

Basically, we must find valid combinations of joins generating tuples answering the query. For enumerating plans using DJ-joins and  $\bowtie$  only we focus on left-deep-tree enumeration [8]. The following plans generate records for the target schema of query  $q1$  we generate following four plans (for improving readability STE is omitted):

$$P1(CT PCP PA) = (CT \bowtie DJ(CT, PCP, company) \text{ as } CTPCP) \bowtie DJ(CTPCP, PA, person)$$



**Figure 4: Candidate plan 5 for query  $q1$  utilizing CJ- and DJ- joins**

$$P2(PCP \ PA \ CT) = (PCP \bowtie DJ(PCP, \ PA, \ person)) \text{ as } PCPPA \bowtie DJ(PCPPA, \ CT, \ company)$$

$$P3(PCP \ CT \ PA) = (PCP \bowtie DJ(PCP, \ CT, \ company)) \text{ as } PCPCT \bowtie DJ(PCPCT, \ PA, \ person)$$

$$P4(PA \ PCP \ CT) = (PA \bowtie DJ(PA, \ PCP, \ person)) \text{ as } PAPCP \bowtie DJ(PAPCP, \ CT, \ company)$$

Plan 1 starts with obtaining CT-relations. Next, for each found company in CT, matching PCP-relations are obtained with a DJ join and joined as CTPCP-relations. Finally, for each CTPCP-relation, matching PA-relations are obtained via another DJ-join and joined as CTPCPPA relations. Plan 2, 3 and 4 are analogous.

For enumerating plans including CJ-joins, first we enumerate possible CJ-joins: Since argument order for CJ-joins is associative, we enumerate combinations. We elect two out of three relationships: [CT PA], [CT PCP], [PCP PA]. The combination [PA CT] is omitted from this list, since no common join predicate exists. There is only one combination for three relationships: [CT PA PCP]. Based on these combinations again we apply a left-deep-tree enumeration to enumerate following plans:

$$P5(CT \ [PCP \ PA]) = CT \bowtie DJ(CT, \ CJ(PCP, \ PA, \ person), \ company)$$

$$P6(PA \ [CT \ PCP]) = PA \bowtie DJ(PA, \ CJ(CT, \ PCP, \ company), \ person)$$

P7([PCP PA] CT) = (CJ(PCP, PA, person) as PCPPA) ⋈ DJ(PCPPA, CT, company)

P8([CT PCP] PA) = (CJ(CT, PCP, company) as CTPCP) ⋈ DJ(CTPCP, PA, person)

P9([CT PCP PA]) = CJ(CT, PCP, PA, company, person)

Plan 5 processes CT-relations first. Next, for each company in CT, [PCP PA]-relations are obtained due a DJ(CT, CJ(PCP,PA) and finally joined to CT PCP-relations. Plan 6, 7 and 8 are analogous. In Plan 9 relations for CT, PCP and PA relationships are joined using a single CJ-function.

### 3 Case Study

As stated in the introduction, results for individual relationships are frequently incomplete and contain “null values”. They are caused by failures of the extractors or object reconciliation techniques which are executed on document collections returned from a SEARCH/TOP operation. However, the SEARCH operator is often non-deterministic in its result set. For example, two searches in Google with identical keywords may return different results. Moreover, the TOP operator makes SEARCH non-deterministic, even if the result is deterministic, if merely the result order is non-deterministic. For that reason, a join operation is not associative anymore. Therefore, different join execution orders will return different results with a different completeness. Depending on how many join predicates are missing, different join orders influence the size of records that finally answer the query. Therefore we focus on determine a query plan that returns a maximal number of such complete records. More formally, given a query  $Q$ , a function *GeneratePlan* generates queries  $q$ , we chose query  $q$  which maximizes the result size of complete records after executing  $q$ .

$$\max_{q \in \text{GeneratePlan}(Q)} |\text{Execute}(q)|$$

In the rest of this Section we study the impact our operators on the result size of complete records answering the query returned by a generated query plan. Our study bases on example query  $q1$  from Figure 2 and investigate effectiveness and efficiency of eight different execution plans for  $q1$ .

#### 3.1 Heuristics for Plan Selection

Following heuristics are based on common optimization heuristics for relational databases and heuristics specific for our join operators on document collections.

*H1: Prefer plans ordered by join predicate selectivity from high to low.* In relational databases selectivity for predicate “p” is the probability that any row from a database satisfies predicate “p”. Operator DJ bases on this principle, in particular on the principle of inclusion. It produces many results if the join predicates from the outer relationship which are likely to be included in the join predicate of the inner relationship. We prefer plans which have a high join predicate selectivity [1, 18].

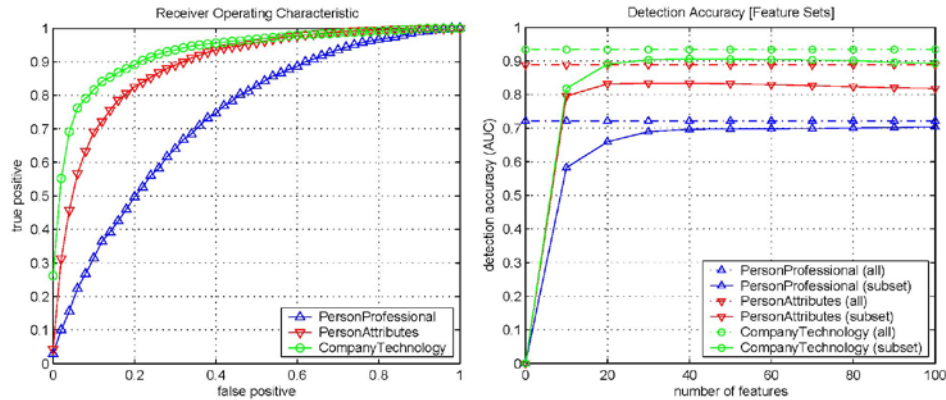
*H2: More documents correlate with larger result sets.* Our intuition is the more documents are in a seed document collection the more likely an extractor is able to extract relevant records.

*H3: For result maximization prefer DJ-joins instead of CJ-joins.* CJ-joins reduce the amount of documents to be considered. We assume that reducing the number of documents will reduce the likelihood to discover further records, relevant for maximizing the result.

### 3.2 Results and Discussion

Table 2 presents results for query  $q1$  and query plans as described in Section 2.5. For each plan we give an abbreviated order of join operations. For example, plan 1-100 and 1-500 follow Figure 3. Brackets denote a CJ-join, for example, plan 5-500 5-100 follows Figure 4. The first eight plans only use DJ-joins while the latter eight plans use DJ and CJ joins. We evaluated each join step separately: Plans start with extracting records  $|R|$  from seed documents  $|D|$ . Next from  $|R|$  we measured the size of distinct join predicates  $|DJP|$ . For example in plan 1-100 for the first join operation we measured the size of distinct companies. Result size for the first join is denoted with  $|R|_{S+1}$  and for the second join with  $|R|_{S+1+2}$ . For each join we measured the selectivity of the join predicate as  $(|R|_{S+1}) / \max(V(P,R), V(P,A))$ , where  $V(P,B)$  is the number of distinct predicates  $P$  in relationship  $A$  and  $V(P,B)$  denotes the size of distinct predicates in relationship  $B$ . Finally, for each plan we measured all documents processed *Overall*  $|D|$  and documents processed to obtain a record *Overall*  $|D| / |R|$ . With respect to our heuristics we could observe following tendencies:

*H1: Join predicate selectivity correlates with result size.* As expected, we could confirm this intuition. The “best” plan 1-100 results starts with the highest selectivity for predicate company when joining relations from relationships CT and PCP. Plan 1-500, ranked 2<sup>nd</sup> among the first eight plans and 5-500 ranked with another plan 1<sup>st</sup> in the second eight plans confirm this heuristic.



**Figure 5: ROC (left) and AUC (right) characteristics for generated keywords**

*H2: More documents do not correlate with higher result size.* We could not confirm H2. Plan 2-500 processed significantly more documents than any other plan but only achieved half the size of records as the best plan. Plan 3-100, 2-100, 4-100 and 4-500 processed even several thousands of documents but either could not return any results or only very few.

*H3: Plans with CJ-Joins are more efficient but reduce result set size.* All plans utilizing CJ-joins (except of 7-500) process significantly fewer documents than plans only using DJ-joins. However, result size for most of these plans is significantly lower than for plans utilizing DJ-joins only.

*Accuracy for generated keywords differs per relationship.* We will briefly discuss accuracy and quality of generated keyword queries. Following the method from [12] we generated keyword queries for each relationship used in query plans for  $q1$  (see Table 2). Our training set based on 11.000 news articles of <http://en.wikinews.org> in the period from 2004 up to 2008 and utilized the Open Calais Extractor to extract records. As shown in Figure 5 (left) detection accuracy depends on the relationship to be learned. The true positive rate varies between 32% and 81% at a false positive level of 10%. Since we are interested in finding feature sets that best describe a particular relationship we investigate the impact of dimension reduction on the overall detection accuracy. Fig. 5 (right) shows that for all three relationships the first 20 features (sorted by relevance) contribute most. The quality of the feature sets reflects the observations from the baseline experiment in Figure 5 (left). Manual observation showed that for *PersonAttributes* and *CompanyTechnology* selected features appear to be largely reasonable, while *PersonProfessional* is described by less meaningful terms, such as corpus artifacts like initials and syllables.

**Concluding remarks: What influences result size?** From our initial study we identify three abilities of the query planer that have a critical impact on the size of the overall result. The first aspect is to *estimate relevant documents* [5,6,17]. Relevant documents contain at least a single record an extractor is expected to identify. Due to the non-deterministic nature of the SEARCH operation, each KEYWORD-SEARCH-

Plan Config			Seed (S)		1st. Join (1)					2nd. Join (2)					Overall	
Plan ID	Order	CJ	D	R	DJP	D	R	R  S+1	Sel	DJP	D	R	R  S+1+2	Sel	D	D  /  R
1-100	CT, PCP, PA	No	100	34	22	220	382	163	7,41	63	630	142	110	1,75	950	8,64
1-500	CT, PCP, PA	No	500	38	24	240	392	140	5,83	43	430	99	82	1,91	1170	14,27
2-500	PCP, PA, CT	No	500	1421	1056	10480	1625	560	0,53	44	440	95	48	1,09	11420	237,92
3-500	PCP, CT, PA	No	500	1370	135	1350	215	68	0,50	30	300	128	31	1,03	2150	69,35
3-100	PCP, CT, PA	No	100	259	21	210	53	15	0,71	9	90	21	3	0,33	3590	1196,67
2-100	PCP, PA, CT	No	100	255	216	2160	479	114	0,53	5	50	18	3	0,60	3150	1050,00
4-500	PA, PCP, CT	No	500	240	238	2360	1662	126	0,53	24	240	31	3	0,13	3100	1033,33
4-100	PA, PCP, CT	No	100	52	51	505	787	46	0,90	12	120	15	0	0,00	1820	--
8-100	CT, (PCP&PA)	Yes	100	35						23	230	45	17	0,74	100	5,88
5-500	(CT&PCP), PA	Yes	350	18						11	110	49	17	1,55	350	20,59
5-100	(CT&PCP), PA	Yes	100	21						15	150	50	14	0,93	100	7,14
8-500	CT, (PCP&PA)	Yes	500	41						29	290	40	11	0,38	500	45,45
7-500	PA, (CT&PCP)	Yes	500	240						234	2265	79	6	0,03	500	83,33
7-100	PA, (CT&PCP)	Yes	100	54						53	505	28	0	0,00	100	--
6-500	(PCP&PA), CT	Yes	450	4						4	40	7	0	0,00	450	--
6-100	(PCP&PA), CT	Yes	100	3						3	30	7	0	0,00	100	--

Intermediate result

Final result

**Table 2: Results for generated plans for query  $q1$**

TOP combination might produce a different set of relevant documents. Second, the query planer needs to *estimate record size of extracted documents*. Relevant documents might differ in their size of extractable records per document for a given relationship. Depending on the quality of the extractor [9] in EXT, some records contained in a relevant document will not be recognized or wrong facts will be extracted. Last, similar to a relational database, the query planer needs to *estimate the selectivity for a predicate in a join-operation*.

## 4 Related Work

We will briefly review related work about query processing in relational- and document collections, quality-driven query planning and text analytics.

**Query planning over document collections.** In the “*extract-then-query-strategy*” structured data is extracted from text data in a static extraction pipeline. Next, extracted records belonging to the same object are merged and fused. Finally, join and aggregation operations are executed on extracted records. Example applications include intranet search [2] or email search. A major disadvantage of this strategy is that a query processor may extract and process many irrelevant documents for a given query. Therefore this strategy is not suitable for business intelligence scenarios, where an OLAP query is executed against document collections obtained at query time. The

*“sample-extract-join-strategy”* by [4] avoids a full extraction over all text sources by selecting relevant documents using keyword based sampling techniques. Their optimizer balances between result recall, precision and execution time according to user specific settings. Plan optimization bases on selection-push-down techniques. More relevant to our work are join implementations, such as Outer Join or Zig-Zag join, proposed in [5]. However, a user needs to balance manually between execution time, recall and precision. For an average analyst this is often infeasible, since no statistics about the “text quality”, such as relationship distribution per document, is known a-priori. Second, extracted relationships are sparse. When joining such data, many “null” values negatively affect data completeness of the final result. Our DJ and CJ address these problems. The relevance ranking approach of NAGA [3] is orthogonal to us. Documents collections identified at query time could be ranked with Naga on top of our system.

**Quality-driven query planning in integrated information systems.** In [7] authors propose data quality-driven query planning techniques. We apply their ideas on data density and data source coverage. Their query planning is executed over a comparable small set of a-priori known data sources. In contrast, in our system, document collections are selected in at query time.

**Relational data base optimization.** Among the large amount of literature on join reordering in relational databases most relevant for us is the work on System-R [8]. Our enumeration in left-deep-join trees for three or more relationships from Section 2.5 bases on this work. More recently, [1] investigated feedback loops for learning correct cardinality estimates for the LEO optimizer. By monitoring queries as they execute, the autonomic optimizer compares the optimizer's estimates with actual cardinalities at each step in a query execution plan, and computes adjustments to its estimates that may be used during future optimizations of similar queries. Our future work may follow these ideas in such a way, that extracted data is used to update relationship cardinality statistics.

**Information extraction and text analytics.** The AVATAR [9] and CIMPLE [10] project describe the extraction plan using an abstract, predicate-based rule language. Such plans offer a wide range of optimization perspectives orthogonal to us. For example, authors in [11] reused cached results from list-based extractors for the same extraction task. Determining appropriate keywords for a structured query based on learning-based strategies has been in [12]. We applied this approach in Section 2.4 for obtaining keywords.

## 5 Summary and Future Work

We investigated effects of re-ordering joins in the context of query planning and query execution over document collections selected at query time. We introduced two join operations, CD and DJ for executing joins over document collections. In a case study we tested heuristics for selecting a plan that maximizes result size. In our study

we identified three critical dimensions for estimating the result size of a generated plan: First, the ability of the planner to produce and estimate the amount of relevant documents answering the query, e.g., by leveraging keyword generation techniques. Next, for a relevant document the query planner needs to estimate the size of extracted records expected to answer the query. That implies estimating the average size of records per relevant document and estimating the precision of the corresponding extractor. Last, the planner needs to estimate the selectivity of join predicates in the plan, e.g., based in observed statistics for “similar” document collections in the past.

Based on results of this study our future work will cover four direction: Existing methods [19,12] to generate keyword queries for a structured query do not consider the context of a query plan. We will investigate how to improve these methods with respect to query planning, e.g., by incorporating context information from intermediate results during query execution, the join order in the plan or additional information about the underlying search engine. Another direction is to develop a solid mathematical model for estimating result size for a generated plan. Currently, several building blocks exist in literature to estimate result size of a single join [17] or to estimate predicate selectivity [1]. By incorporating these and further building blocks we will develop a cost model for estimating result size of a more complex query plan, e.g., a plan involving multiple joins. Further, given the sheer amount of different document collections on the web, a critical research task is to gather and update compact statistics that could be reused for multiple document collections. Finally, based on a logical execution plan; we will compile a physical execution plan towards a parallel execution model.

## References

1. Markl, V., Lohman, G. M., and Raman, V. 2003. LEO: An autonomic query optimizer for DB2. *IBM Syst. J.* 42, 1 (Jan. 2003), 98-106.
2. Zhu, H., Raghavan, S., Vaithyanathan, S., and Löser, A. 2007. Navigating the intranet with high precision. WWW '07. ACM, New York, NY, 491-500.
3. Kasneci, G., Suchanek, F. M., Ifrim, G., Elbassuoni, S., Ramanath, M., and Weikum, G. 2008. NAGA: harvesting, searching and ranking knowledge. SIGMOD '08. ACM, New York, NY, 1285-1288.
4. Jain, A., Doan, A., and Gravano, L. 2008. Optimizing SQL Queries over Document collections. ICDE. IEEE Computer Society, Washington, DC, 636-645.
5. Jain, A. and Srivastava, D. 2009. Exploring a Few Good Tuples from Document collections. ICDE. IEEE Computer Society, Washington, DC, 616-627.
6. Jain, A., Ipeirotis, P. G., Doan, A., and Gravano, L. 2009. Join Optimization of Information Extraction Output: Quality Matters! ICDE. IEEE Computer Society, Washington, DC, 186-197.
7. Naumann, F., Leser, U., and Freytag, J. C. 1999. Quality-driven Integration of Heterogeneous Information Systems. Very Large Data Bases. Morgan Kaufmann Publishers, San Francisco, CA, 447-458.

8. Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G. 1979. Access path selection in a relational database management system. *SIGMOD '79*. ACM, New York, NY, 23-34.
9. Kandogan, E., Krishnamurthy, R., Raghavan, S., Vaithyanathan, S., and Zhu, H. 2006. Avatar semantic search: a database approach to information retrieval. *SIGMOD '06*. ACM, New York, NY, 790-792.
10. Shen, W., DeRose, P., McCann, R., Doan, A., and Ramakrishnan, R. 2008. Toward best-effort information extraction. *SIGMOD '08*. ACM, New York, NY, 1031-1042.
11. Krishnamurthy, R., Li, Y., Raghavan, S., Reiss, F., Vaithyanathan, S., and Zhu, H. 2009. System T: a system for declarative information extraction. *SIGMOD Rec.* 37, 4 (Mar. 2009), 7-13.
12. Agichtein, E. and Gravano, L. 2003. QXtract: a building block for efficient information extraction from document collections. *SIGMOD '03*. ACM, New York, NY, 663-663.
13. Jain, A. and Ipeirotis, P. G. 2009. A quality-aware optimizer for information extraction. *ACM Trans. Database Syst.* 34, 1 (Apr. 2009), 1-48.
14. Galhardas, H., Florescu, D., Shasha, D., Simon, E., and Saita, C. 2001. Declarative Data Cleaning: Language, Model, and Algorithms. *Very Large Data Bases*. Morgan Kaufmann Publishers, San Francisco, CA, 371-380.
15. Vapnik, V.: *Statistical Learning Theory*. Wiley Interscience Publication, New York (1998)
16. Joachims, T.: *Making large-scale support vector machine learning practical*. MIT press, Cambridge, MA. (1989)
17. Ipeirotis, P. G., Agichtein, E., Jain, P., and Gravano, L. 2006. To search or to crawl?: towards a query optimizer for text-centric tasks. *SIGMOD '06*. ACM, New York, NY
18. Garcia-Molina, H. Ullmann, J.D., Widom, J., *Database Systems: The Complete Book*. Prentice Hall (2008)
19. Liu J., X. Dong, X., Halevy, A. Answering Structured Queries on Unstructured Data. *WebDB 2006*.
20. Xin Dong, Alon Y. Halevy, Jayant Madhavan: Reference Reconciliation in Complex Information Spaces. *SIGMOD Conference 2005*: 85-96

### **Appendix: Implementing the KEYWORD-Operator.**

We give a brief summary on our implementation of the keyword operator. Ideally, we could sample relevant documents for extracting a complete record answering a structured query, or documents providing records for multiple relationships for answering a query. Estimating the right keywords to obtain such documents is a difficult task. Our approach is based on a machine learning technique that is proposed in [12]. The basic idea is to identify relevant documents for an extractor a-priori on a training set and learn common features describing these documents. These features represent our keyword queries. To this end, extractors relevant for answering a structured query were applied on a set of training documents. In order to obtain a two-class setup, documents for which a relationship has been identified by the corresponding extractor were assigned a positive label whereas the remaining part of the training corpus was labeled negative. With a labeled data set at hand we applied a Support Vector Machine (SVM) [16] to find a hyperplane that separates both classes such that their margin is maximal. Finally, informative features (i.e. features that have a significant contribution on the width of the hyperplane) were selected to form the set of keywords which describes the relationship in question.