# Processing Ad-Hoc Joins on Mobile Devices

Eric Lo[1], Nikos Mamoulis[1], David W. Cheung[1], Wai Shing Ho[1], and Panos Kalnis[2]

[1] Department of Computer Science and Information Systems, The University of Hong Kong
[2] Department of Computer Science, National University of Singapore

**Abstract.** Mobile devices are capable of retrieving and processing data from remote databases. In a wireless data transmission environment, users are typically charged by the size of transferred data, rather than the amount of time they stay connected. We propose algorithms that join information from non-collaborative remote databases on mobile devices. Our methods minimize the data transferred during the join process, by also considering the limitations of mobile devices. Experimental results show that our approach can perform join processing on mobile devices effectively.

## 1 Introduction

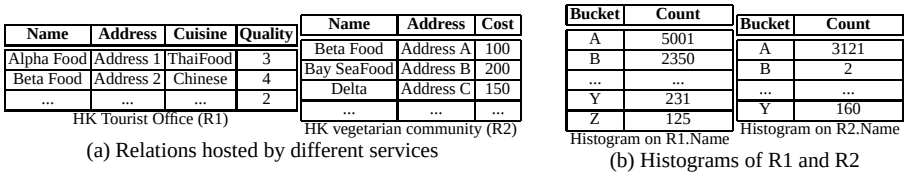
Recently, mobile devices such as Personal Data Assistants (PDAs) and mobile phones are fast emerging as components of our daily life. Many applications of mobile devices access data on remote servers through a wireless network. For example, some people may ask for stock ticks through their PDAs wirelessly. In addition to querying a single remote information source, some mobile users may wish to combine information from different remote databases. Consider for example a vegetarian, who visits Hong Kong and looks for some restaurants recommended by both the HK tourist office and the HK vegetarian community. He may issue a query that joins the contents of two relations in Figure 1(a) hosted by the two different services:

```
SELECT R1.Name, R1.Address, R2.Cost FROM R1, R2 WHERE R1.Name = R2.Name
```

This query is an example of an *ad-hoc* join (where the join key is "Name"), with few results compared to the large volume of input data, and the joined relations are located at different non-collaborative servers. Since the target relations are non-collaborative and users may pose queries on arbitrary sets of data sources, distributed mediator systems like HERMES [1] may not be applicable. Also, remote servers would only accept simple input like selection queries, expressed in SQL through public interfaces, like Web services, etc. Thus traditional distributed query processing techniques that involve shipments of data structures like semijoin [2] and bloomjoin [3] are not applicable.

Downloading both relations entirely to perform a join operation on the mobile device may not be the best solution. First, some devices (e.g., mobile phones) do not have enough memory to hold the large volume of data locally. Second, the network traffic or QoS of various wireless data networking technologies such as CDMA [4], IEEE 802.11 [5], WAP (`www.wapforum.org`) and 3G (`www.nokia.com/3g/index.html`) are all strongly dependent on many factors such as the network workloads and the availability of various network stations. As a result, mobile service providers often charge their

customers in terms of the amount of data transferred, rather than the amount of time they stay connected to the service or the query processing cost on the data host. Therefore, the cost of joining relations from different remote databases does not involve the traditional I/O and CPU costs, but the *transfer cost*.



| Name | Address | Cuisine | Quality |
|---|---|---|---|
| Alpha Food | Address 1 | ThaiFood | 3 |
| Beta Food | Address 2 | Chinese | 4 |
| ... | ... | ... | 2 |

HK Tourist Office (R1)

| Name | Address | Cost |
|---|---|---|
| Beta Food | Address A | 100 |
| Bay SeaFood | Address B | 200 |
| Delta | Address C | 150 |
| ... | ... | ... |

HK vegetarian community (R2)

(a) Relations hosted by different services

| Bucket | Count |
|---|---|
| A | 5001 |
| B | 2350 |
| ... | ... |
| Y | 231 |
| Z | 125 |

Histogram on R1.Name

| Bucket | Count |
|---|---|
| A | 3121 |
| B | 2 |
| ... | ... |
| Y | 160 |

Histogram on R2.Name

(b) Histograms of R1 and R2

**Fig. 1.** Examples relations and histograms

In this paper, we study the evaluation of joins on mobile devices, taking those issues (independent data sources, limited memory size and transfer-cost based optimization) under consideration. We propose *RAMJ* (*R*ecursive and *A*daptive *M*obile *J*oin) algorithm that runs on a mobile device and joins two relations located on non-collaborative remote servers. RAMJ partitions the data space and downloads *statistical* information from the servers for each partition, before downloading the data to be joined. Based on the statistics of a partition, RAMJ may *adaptively* choose to download data that fall into that partition and join them using a join technique, or to apply RAMJ *recursively* for retrieving more refined statistics there.

The rest of the paper is organized as follows. Section 2 describes various basic approaches and RAMJ for evaluating equi-join queries on mobile devices. Section 3 describes how RAMJ can be applied on other query types. Section 4 reports the experiment result. Section 5 describes the related work and we conclude in Section 6 with some directions for future research.

## 2 Processing of Ad-Hoc Equi-Joins

### 2.1 Basic approaches

**A late-projection strategy**. In highly selective joins, most tuples fail to contribute to the result. One naive way to reduce the transfer cost and memory requirements of the join is to download and join the *distinct values of join keys* only. Subsequently, only tuples belonging to the join result entails downloading the rest of non-key attributes from both sites. The late projection strategy avoids downloading the non-key attributes which do not contribute to the result, and can be applied to all types of join queries. Therefore we adopt this strategy in the rest of the paper.

**Block-merge join (BMJ)**. The late-projection technique is better than downloading all relations and joining them locally, but it is still insufficient when the results of the projections of join keys do not fit into the memory. A straightforward method to evaluate the equi-join with limited memory is sort-merge join, by assigning the sorting part to the

servers. Then, only one block from each server is required to be in memory and computational cost on the client is saved by forcing the servers to sort the data. The idea is to download one block of *ordered* join keys from each relation (using the `ORDER BY` SQL statement) and merge them, until one block is exhausted. Then a new block is retrieved from the corresponding server and the process continues until one relation is scanned.

**Ship-data-as-queries join (SDAQ)**. If two target relations R1 and R2 have large size difference (e.g., R1 $\ll$ R2), the join can be evaluated by downloading *all* join keys of R1 to the device and sending each of them as a selection query to R2. Before that, two SQL queries are sent to count the number of tuples on the join attributes in each server, in order to identify the relation with smaller cardinality. This method resembles the semijoin [2] approach in System R* [6], with the only difference that the join keys are expressed as queries and shipped through the mobile devices indirectly. If the set of join keys of the smaller relation does not fit into the device, SDAQ can be implemented by downloading the join keys from R1 and sending them to R2, in a block-wise fashion, like the block-merge join.

### 2.2 RAMJ: A recursive and adaptive approach

BMJ requires that all data from both relations are downloaded. SDAQ, on the other hand, can only do better if the sizes of the relations differ much. In fact, when the data distributions of the join keys are different, there may be empty key value ranges in one relation that are densely populated in the other. Such knowledge may help avoid downloading data in these ranges, as we know that they do not participate in the join result. Figure 1(b) shows two histograms summarizing the "Name" attribute of our example relations. If the value ranges which are empty in one relation (e.g., bucket "Z" of R2), they are pruned and their contents are not downloaded from the other relation (e.g., R1).

In general, remote services accept *selection* queries only, i.e., we cannot create any temporary relation on remote servers to facilitate the collection of statistics. In view of this, the intuition behind our method is to apply some cheap queries first, in order to obtain some information about the distribution of the data values of join attributes in both datasets. A simple way to construct a histogram on an attribute is to pose an aggregate query. For this, we can use the `SUBSTRING` function and `ROUND` function. An example SQL statement that returns the left histogram of Figure 1(a) (grouped by first character) is given below. Notice that the `HAVING` clause avoids fetching value ranges with no data to reduce data transferred.

```
SELECT SUBSTRING(Name,1,1) AS Bucket, COUNT(Name) AS Count FROM R1
GROUP BY SUBSTRING(Name,1,1) HAVING COUNT(Name) > 0
```

Retrieving and comparing histograms prior to actual data contents not only reduces the join space; in fact, histogram buckets are also employed as the basic unit of our recursive solution. Essentially, we divide the problem into $G$ smaller ones. Subsequently, each smaller join can be processed by a different method. Thus, for each pair $\langle \alpha_i, \beta_i \rangle$ of buckets (where $\alpha_i$ comes from R1 and $\beta_i$ comes from R2), we consider *one* of the following actions according to the bucket size:

**Direct join**. This method applies the merge-join algorithm for the current partition. Essentially, the data in the partition are downloaded from R1 and R2 and then joined on the mobile device, using BMJ.

**Ship-join-keys join**. This method shares the same idea of ship-data-as-queries join that downloads the join keys of the *smallest* partition and sends them as selection queries to the other service. It can be cheaper than direct join, if the data distributions in the two buckets are very different.

**Recursive partitioning**. This method breaks a partition into more refined ones and requests histograms for them. Essentially, it further breaks a bucket into smaller ones, hoping that some smaller buckets will be eliminated or cheap ship-join-keys joins will be applied on them. However, if partitions' data are uniform, it could be more expensive than direct join, given the overhead for retrieving the histograms.

To illustrate, consider the level-1 histograms of Figure 1(b) again. After obtaining the histograms, bucket Z of R1 can be pruned immediately, since the corresponding one of R2 is empty. For bucket Y, as the number of join keys that fall into this bucket for both relations is quite small and balanced, downloading them and applying direct join may be the best choice because (i) the ship-join-keys join will have a larger overhead (for downloading the data from R2, sending them to R1 as queries and then retrieving the results) and (ii) the overhead of repartitioning and retrieving further statistics may not pay-off for this bucket size. For bucket A, we may want to apply recursive partitioning, hoping that data distribution in the refined buckets will be more skewed, helping to prune the join space. The finer histogram of bucket A may consist of 26 sub-buckets ("AA","AB",...,"AZ") by issuing a SQL statement that group by the first two characters. Finally, for bucket B, ship-join-keys is arguably the best method since only two join keys from R2 will be downloaded and shipped to R1.

### 2.3 The cost model

We provide formulae which estimate the cost of each of the three potential actions that RAMJ may choose. Our formulae are parametric to the communication cost of the wireless network services. In general, $T(B)$ bytes are transmitted through a network for $B$ bytes of data because some overhead are spent in each transfer unit (e.g., MTU in TCP/IP). Let $\lambda_1$ and $\lambda_2$ be the per-byte transfer cost (e.g., in dollars) for accessing R1 and R2 respectively. Sending a selection query $Q$ to a server needs $T(B_{SQL} + B_{key})$ bytes, where $B_{SQL}$ denotes the size of a SQL statement (different query types have different $B_{SQL}$ values) and $B_{key}$ reflects the cost of sending the key value that defines the selection. Due to space constraints, details of derivation of the following cost equations are omitted and interested readers are referred to [7].

**Direct join**. Let $\alpha_i$ and $\beta_i$ be the $i$-th histogram bucket summarizing the same data region of R1 and R2 respectively. Further, let $|\alpha_i|$ and $|\beta_i|$ be the number of tuples represented by $\alpha_i$ and $\beta_i$. The total transfer cost of downloading all key values represented by $\alpha_i$ and $\beta_i$ and joining them on the mobile device is:

$$C_1(\alpha_i, \beta_i) = (\lambda_1 + \lambda_2)T(B_{SQL} + B_{key}) + \lambda_1 T(|\alpha_i| B_{key}) + \lambda_2 T(|\beta_i| B_{key}) \qquad (1)$$

The first term is the transfer cost of two outgoing queries and the last two terms are the transfer cost of downloading the data contents represented by $\alpha_i$ and $\beta_i$.

**Ship-join-keys join**. Assuming $|\alpha_i| \leq |\beta_i|$, the total transfer cost $C_2$ of joining $\alpha_i$ and $\beta_i$ using ship-join-keys join is:

$$C_2(\alpha_i, \beta_i) = \lambda_1(T(B_{SQL} + B_{key}) + T(|\alpha_i|B_{key})) + \lambda_2(T(B_{SQL} + 2|\alpha_i|B_{key})) \quad (2)$$

The first term is the transfer cost of the selection query sent to R1 and its query result. The second term is the transfer cost of the selection query sent to R2 for checking existence of R1 keys and its query result.

**Recursive partitioning**. The cost $C_3$ of applying recursive partitioning on $\alpha_i$ and $\beta_i$ composes of three sub-costs:

1. $C_h(G, R1)$: The cost of retrieving $G$ bucket-count pairs $\lambda_1(T(G(B_{key} + B_{int})))$ that refine bucket $\alpha_i$ by submitting an aggregate `COUNT` query $\lambda_1(T(B_{SQL} + B_{key}))$ to R1.

2. $C_h(G, R2)$: The cost of retrieving $G$-bucket-count pairs from R2:
$\lambda_2(T(G(B_{key} + B_{int})))+ \lambda_2(T(B_{SQL} + B_{key}))$

3. Let $\alpha_{i,j}$ and $\beta_{i,j}$ ($j = 1 \cdots G$) be the set of sub-buckets of $\alpha_i$ and $\beta_i$ respectively. After downloading the refined histograms, each pair of sub-buckets are examined recursively to determine the next optimal transfer cost action. Therefore, the last sub-cost of $C_3$ is: $C_{RP}(\alpha_i, \beta_i) = \sum_{j=1}^{G} \min_{l \in \{1,2,3\}} C_l(\alpha_{i,j}, \beta_{i,j})$

Since we have no knowledge on how the data are distributed in the sub-buckets, we cannot directly determine the last sub-cost $C_{RP}(\alpha_i, \beta_i)$ (observe that it contains a $C_3$ component), we introduce two different models to estimate this last sub-cost.

*Optimistic Estimation.* When the join is highly-selective, the data distribution of the two datasets is quite different; thus, one way to deal with the unknown component is to optimistically assume all buckets are pruned in next level:

$$C_3(\alpha_i, \beta_i) = C_h(G, R1) + C_h(G, R2) \quad (3)$$

This model tends to obtain finer statistics in each level. It is a simple approach with small computation demand and shown to be quite effective in the experiment section.

*Estimation by Linear Interpolation.* This model estimates the last sub-cost $C_{RP}(\alpha_i, \beta_i)$ more accurately, but with higher computation cost on the client device. The idea is to exploit the histograms in the coarser level to speculate the local data distribution in the next level. To estimate the last sub-cost $C_{RP}(\alpha_i, \beta_i)$, the counts of sub-buckets $\alpha_{i,j}$ and $\beta_{i,j}$ have to be determined. Take $\alpha_i$ as an example. We propose to distribute the count of $\alpha_i$ to its sub-buckets $\alpha_{i,j}$ by linear interpolation. Adjacent buckets are selected to be the interpolation points because they are more related to local skew. Thus, to estimate the data distribution in the buckets of the finer level, only two adjacent buckets ($\alpha_{i-1}$ and $\alpha_{i+1}$) of the current bucket ($\alpha_i$) at the current level are selected. The sub-buckets are then weighted by the following formulae:

$$\text{When G is even} \begin{cases} W_{\alpha_{i,j}} = |\alpha_i| + (|\alpha_{i+1}| - |\alpha_i|)(2(j - G/2) - 1)/2G \text{ if } j > G/2 \\ W_{\alpha_{i,j}} = |\alpha_i| + (|\alpha_{i-1}| - |\alpha_i|)(2(G/2 - j) + 1)/2G \text{ if } j \leq G/2 \end{cases}$$

$$\text{When G is odd} \begin{cases} W_{\alpha_{i,j}} = |\alpha_i| + (|\alpha_{i+1}| - |\alpha_i|)(j - \lceil G/2 \rceil)/G \text{ if } j > G/2 \\ W_{\alpha_{i,j}} = |\alpha_i| + (|\alpha_{i-1}| - |\alpha_i|)(\lceil G/2 \rceil - j)/G \text{ if } j \leq G/2 \end{cases}$$

After weighting, the count of bucket $\alpha_i$ is distributed to the sub-buckets according to the weights, i.e. $|\alpha_{i,j}| = |\alpha_i| W_{i,j} / \sum_{j=1}^{G} W_{i,j}$. Now, for each bucket pair of the next

level, the cost of direct join ($C_1(\alpha_{i,j}, \beta_{i,j})$) and ship-join-keys join ($C_2(\alpha_{i,j}, \beta_{i,j})$) can be determined using the estimates, according to equations 1 and 2. However, the sub-cost $C_{RP}(\alpha_{i,j}, \beta_{i,j})$ cannot be estimated, as those sub-buckets may recursively be partitioned and there is no more information to further estimate their recursive actions. Fortunately, the lemma below provides an appropriate upper-bound for our estimation.

**Lemma 1.** *Let $\alpha_i$ and $\beta_i$ be the i-th bucket of the $G$-bucket equi-width histograms downloaded from R1 and R2 respectively. The cost $C_{RP}(\alpha_i, \beta_i)$ is bounded by the inequality: $C_{RP}(\alpha_i, \beta_i) \leq \sum_{j=1}^{G} min(C_1(\alpha_{i,j}, \beta_{i,j}), C_2(\alpha_{i,j}, \beta_{i,j}))$* □

A proof of the lemma can be found in [7]. Using Lemma 1, we can estimate an upper bound for $C_{RP}(\alpha_i, \beta_i)$. Hence, $C_3$ becomes:

$$C_3(\alpha_i, \beta_i) = C_h(G, R1) + C_h(G, R2) + \sum_{j=1}^{G} min(C_1(\alpha_{i,j}, \beta_{i,j}), C_2(\alpha_{i,j}, \beta_{i,j})) \quad (4)$$

### 2.4 The RAMJ algorithm

Although the action and the cost estimation of every bucket are determined in runtime, RAMJ defers all bucket actions and execute them in batches according to their action types. Thus a constant number of queries is sent out in each level and reduces the packet headers overhead, since multiple requests are "compressed" in a single statement.

**Algorithm RAMJ(**$R1, R2, G, L$**)**
/* $G$ is the partition granuality; $L$ is the list of buckets currently served*/
1. $H_{R1}$ := BuildHist($R1, G, L$);
2. $H_{R2}$ := BuildHist($R2, G, L$);
3. For each bucket pair $\langle \alpha_i, \beta_i \rangle, \alpha_i \in H_{R1}, \beta_i \in H_{R2}, \alpha_i = \beta_i$
4. $C_{min} := \min(C_1(\alpha_i, \beta_i), C_2(\alpha_i, \beta_i), C_3(\alpha_i, \beta_i))$;
5. If $C_{min} = C_1(\alpha_i, \beta_i)$, add current bucket to $L_{C1}$;
6. If $C_{min} = C_2(\alpha_i, \beta_i)$, add current bucket to $L_{C2}$;
7. If $C_{min} = C_3(\alpha_i, \beta_i)$, add current bucket to $L_{C3}$;
8. If $L_{C1} \neq \emptyset$, execute direct join $\forall$ buckets $\in L_{C1}$;
9. If $L_{C2} \neq \emptyset$, execute ship-join-keys join $\forall$ buckets $\in L_{C2}$;
10. If $L_{C3} \neq \emptyset$, RAMJ($R1, R2, G, L_{C3}$);

The above shows the RAMJ algorithm. RAMJ follows the bucket-wise approach and it is recursive; given two remote relations R1 and R2, the algorithm first draws two $G$-bucket equi-width histograms $H_{R1}$ and $H_{R2}$ that summarize the distribution of join key in R1 and R2 (Lines 1–2). In `BuildHist`, $L$ is the set of buckets from the previous level that are currently refined as a batch. When RAMJ is called for the first time, $L$ corresponds to a single bucket that covers the whole domain of the join attribute. For each refined bucket range that exists in both relations, RAMJ employs the cost model to estimate the cost of each of the three potential actions for the current bucket. The action of each bucket is deferred until the costs of all buckets have been estimated. Finally, Lines 8–10 execute each of the three actions (if applicable) for the corresponding set of bucket ranges, as a batch. We note here, that if the memory of the device is limited, the device may not be able to support action deferring or even direct join or ship-join-keys join for a specific bucket (or set of buckets). In this case, recursive partitioning is directly used without applying the cost model to choose action.

## 3 Processing of Other Join Queries

We have discussed RAMJ for the case where there is a single join attribute. In case of multiple attributes, RAMJ can be simply adapted by retrieving multidimensional histograms and adjusting the cost formulae accordingly.

Sometimes, users may apply some selection constraints on the joined relations. Constraints are usually expressed by (allowed) value ranges on the attribute domain. Consider the example query in the introduction, assume that the user is now only interested in restaurants with cost lower than $20 recommended by both HK tourist office and vegetarian community:

`SELECT R1.Name, R1.Address, R2.Cost FROM R1,R2 WHERE R1.Name=R2.Name` *and R2.Cost*$<$*20*

RAMJ can efficiently process such queries by "pushing" the selections as early as possible, following the common optimization policy of database systems. The selections are sent together with the histogram requests to avoid including disqualified tuples in the bucket counts. Therefore, only tuples that satisfy all conditions are summarized and the adaptivity of our algorithm is not affected.

RAMJ is also useful for iceberg semijoin queries. As an example, consider the query "find all restaurants in R1 which are recommended by at least 10 users in a discussion group R2". Such queries require joining two remote relations and retrieving only the tuples in one (e.g., R1) that join with at least $t$ tuples in the other (e.g., R2). The result is usually small, making it useful to the mobile user. RAMJ can easily handle such queries by modifying the aggregate `COUNT` query in `BuildHist` procedure; buckets with count less than the threshold $t$ are not included in the histogram. This can be achieved by modified the `HAVING` predicate from "`HAVING COUNT` (Attribute) $> 0$" to "`HAVING COUNT` (Attribute) $> t$". As a result, large parts of the data space can be pruned by the cardinality constraint early.

## 4 Performance Experiments and Results

To evaluate the effectiveness of the proposed RAMJ algorithm, we have conducted experiments on both real and synthetic datasets. The remote datasets resided on two Oracle 8i servers. We have implemented RAMJ on mentioned cost models, i.e., optimistic (RAMJ-OPT) and linear interpolation (RAMJ-LI). In addition, we also report results for an optimal version of RAMJ (OPTIMAL) that pre-fetches the next-level histograms to determine the "optimal" action for each bucket. This version is only of theoretical interest, and serves as a lower bound for our cost model. The mobile device is simulated by a Pentium PC with 8MB memory available to RAMJ. We compare all versions of RAMJ with the basic approaches, i.e., block-merge join (BMJ) and ship-data-as-query join (SDAQ). For fairness, all implemented algorithms employ the late-projection strategy (i.e., joining the set of join attributes only).

The real data set contains information of 152K restaurants crawled from `restaurantrow.com`. We split the restaurant data into different (overlapping) sets according to the cuisine they offered. We then joined these sets to retrieve restaurants offering more than one type of cuisine. We present an illustrative experiment that joins

| Algorithm | Real Data | | NegExp-Gaussian | | Zipf-Gaussian | | Zipf-NegExp | |
|---|---|---|---|---|---|---|---|---|
| | *Transferred (Bytes)* | *No. of joined keys* | *Transferred (Bytes)* | *No. of joined keys* | *Transferred (Bytes)* | *No. of joined keys* | *Transferred (Bytes)* | *No. of joined keys* |
| BMJ | 266.22K | 163 | 80116 | 420 | 80124 | 184 | 80120 | 1148 |
| SDAQ | 180.15K | | 181944 | | 139728 | | 143580 | |
| RAMJ-OPT | 116.67K | | 48654 | | 35056 | | 77114 | |
| RAMJ-LI | n/a | | 40956 | | 25680 | | 67092 | |
| OPTIMAL | 66.2K | | 35100 | | 21436 | | 34848 | |

**Table 1.** Experimental results

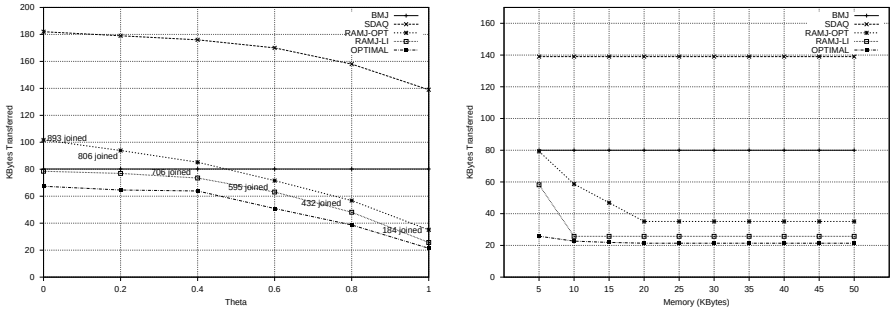
relation *steaks* (4573 tuples) with *vegetarian* (2098 tuples) to identify the restaurants that offer both steak and vegetarian dishes. Experiments joining different cuisine combinations have similar results. Table 1 shows that the data transferred by RAMJ-OPT are only 44% and 65% compared to BMJ and SDAQ respectively. SDAQ transferred less bytes than BMJ because the size differences between the two input relations of the restaurant datasets are significant. RAMJ-OPT is more close to OPTIMAL than both BMJ and SDAQ because of its adaptivity. RAMJ-LI was not tested on string data joins since the distribution of characters is independent at different levels.

Next, we study the performance of our algorithms on synthetic data, under different settings. In particular, we studied the amount of data transferred on joining relations in different data distribution by RAMJ. Each input relation consists of 10,000 tuples of integers with domain size 100,000, and $G$ is set to 20.

**Overall performance**. To model the real scenario of ad-hoc joining, we generated data with 3 different distributions: Gaussian, Negative Exponential (NegExp) and Zipf (with skew parameter $\theta = 1$). Table 1 shows that RAMJ outperforms BMJ and SDAQ even when the data have similar distribution (e.g., when joining Negative Exponential data with Zipf data). RAMJ-LI is better than RAMJ-OPT, and closer to OPTIMAL, because RAMJ-LI employs a more accurate cost model.

**The impact of data skew**. Since RAMJ is designed for highly selective join, we study how data skew affects the performance of RAMJ. We generated 6 relations with different Zipf distribution by varying the skewness ($\theta$) from 0 (uniform distribution) to 1. Each of them was joined with the Gaussian data relation. Figure 2(a) shows that the total number of bytes transferred by RAMJ decreases when the join selectivity ($\theta$) increases. It is because when the data distribution is skewed, parts of the search space are pruned and for some buckets cheap ship-join-keys joins are performed, hence, the histograms retrieved by RAMJ pay-off. BMJ transfers a constant amount of bytes because it downloads all tuples, independent of the data skew. If the data are near-uniform, BMJ outperforms RAMJ-OPT because RAMJ-OPT optimistically retrieves more statistics unsuccessfully hoping to prune the search space. On the other hand, RAMJ-LI outperforms BMJ and SDAQ in all cases because it is less sensitive to data skew.

**The impact of memory size**. We evaluate how memory of mobile devices affects the transfer cost. Figure 2(b) shows the performance of joining the Gaussian and Zipf ($\theta$=1) distributions by RAMJ again, under different memory settings. As expected, the transfer cost increases when the memory is very limited (less than 20K). It is because many buckets cannot execute their optimal actions, but need to apply recursive partitioning




(a) Varying s (b) Varying m

**Fig. 2.** Synthetic data experiments

if those actions cannot be performed with the available memory. Note that the transfer cost stabilizes to a constant when the memory is larger than 20K memory. This figure shows that only a small memory on the mobile device suffices for RAMJ to perform the speculated optimal actions.

**Summary of other experiments**. We also ran RAMJ:
i) in different granularities, results show that the performances of RAMJ is unaffected by $G$, if $G$ is not very small ($G < 5$).
ii) in different input relation sizes, which draw similar conclusions as the experiments we presented.
iii) on another real dataset, the DBLP bibliography, results show that RAMJ-OPT transferred 66% of BMJ and 69% of SDAQ respectively.

Readers that interest on the detail of these experiment results and the query processing time of RAMJ (all finished with seconds) are referred to [7].

## 5 Related Work

We are aware of no work in the area of transfer-cost based query optimization (measured in dollars/byte) in the context of join queries on mobile devices. Nonetheless, query processing in a decentralized and distributed environment has been studied extensively in the past two decades [8, 2, 6]. Transfer cost optimization is discussed in some of these works. However, the traditional distributed database model assumes cooperative servers (in a trusted environment), which is not the case in our problem. Notice that, although we have adapted the semijoin [8] approach for this problem, its cost is higher than the other methods in practice.

Our work is also related to distributed query processing using mediators (e.g., [1]). Nevertheless we study queries that combine information from ad-hoc services, which are not supported by mediators. Join processing on mobile devices considering transfer-cost minimization has been studied in [9, 10]. In [9] the mobile clients are distributed database nodes (pairs of nodes are collaborative in nature). On the other hand, [10] applies similar techniques with this paper, but considers spatial joins only.

## 6 Conclusions and Future Work

Emerging mobile devices such as PDAs and mobile phones are creating opportunities for users to access information from anywhere and at any time. However, mobile devices have some limitations that make traditional query processing technology inappropriate. In this paper, we studied join queries on remote sources which are non-collaborative and are not logically connected via mediators. We addressed two issues: (i) the limited resources of the device and (ii) the fact that users are charged by the amount of transferred information, rather than the processing cost on the servers. Indeed, users are typically willing to sacrifice a few seconds in order to minimize the query cost in dollars. Furthermore, we proposed RAMJ, an adaptive algorithm that recursively partitions the data space and retrieves statistics that dynamically optimize equi-join queries. We also discussed how RAMJ can be extended to answer other types of joins including iceberg semijoins and constrained equi-joins. Finally, we evaluated RAMJ on several synthetic and real datasets. Our experiments reveal that RAMJ can outperform the basic approaches for a wide range of highly-selective joins.

RAMJ evaluates the basic equi-join efficiently. Indeed, RAMJ provides a basic framework to evaluate many highly-selective query types that may also involve joining (e.g., evaluating skyline and top-$k$ query over a set of attributes that resided in different non-collaborative servers). In future, we will extend our approach for multi-way join queries, by utilizing the histograms to estimate the size of intermediate join results.

## References


1. Adali, S., Candan, K.S., Papakonstantinou, Y., Subrahmanian, V.S.: Query caching and optimization in distributed mediator systems. In: Proc. of ACM SIGMOD. (1996) 137–148
2. Bernstein, P.A., Goodman, N.: Power of natural semijoin. SIAM Journal of Computing **10** (1981) 751–771
3. Mullin, J.K.: Optimal semijoins for distributed database systems. IEEE Tran. on Software Engineering **16** (1990) 558–560
4. Knisely, D.N., Kumar, S., Laha, S., Nanda, S.: Evolution of wireless data services: IS-95 to CDMA2000. IEEE Comm. Magazine (1998) 140–149
5. Kapp, S.: 802.11: Leaving the wire behind. IEEE Internet Computing **6** (2002)
6. Mackert, L.F., Lohman, G.M.: R* optimizer validation and performance evaluation for distributed queries. In: Proc. of VLDB. (1986) 149–159
7. Lo, E., Mamoulis, N., Cheung, D.W., Ho, W.S., Kalnis, P.: Processing ad-hoc joins on mobile devices. Technical report, The University of Hong Kong (2003) Available at http://www.csis.hku.hk/~dbgroup/techreport.
8. Bernstein, P.A., Chiu, D.M.W.: Using semi-joins to solve relational queries. Journal of the ACM (JACM) **28** (1981) 25–40
9. Lee, C.H., Chen, M.S.: Processing distributed mobile queries with interleaved remote mobile joins. IEEE Tran. on Computers **51** (2002) 1182–1195
10. Mamoulis, N., Kalnis, P., Bakiras, S., Li, X.: Optimization of spatial joins on mobile devices. In: Proc. of SSTD. (2003)