A Vision for Autonomous Data Agent Collaboration: From Query-by-Integration to Query-by-Collaboration

Timo Eckmann
Technical University of Darmstadt
Darmstadt, Germany
timo.eckmann@tu-darmstadt.de

Carsten Binnig
Technical University of Darmstadt & hessian.AI & DFKI
Darmstadt, Germany
carsten.binnig@tu-darmstadt.de

Abstract

In this paper, we present a vision for autonomous data agent collaboration, where complex queries are no longer executed through a single centralized query plan, but instead resolved through a dynamic protocol among intelligent agents—each serving as an autonomous interface to a distinct data source. This decentralized approach eliminates the need for traditional data integration by allowing agents to reason locally while collaboratively contributing to global query results. As a core contribution toward realizing this vision, we introduce key design principles for such systems and outline the foundations of a Query Collaboration Protocol (QCP) to enable decentralized, autonomous query answering across independent and potentially noisy data sources. In our initial experiments with a prototype system, `QCP-DB`, we demonstrate that it can answer complex queries over heterogeneous databases with accuracy comparable to that of fully integrated centralized systems—yet without incurring the cost and complexity of data integration.

1 Introduction

Data is Abundant, but Making Sense of It is a Bottleneck. We live in an era where data about nearly every aspect of our lives is generated in digital, machine-readable form. However, despite this abundance, the true challenge lies not in data availability but in making sense of it—accessing, integrating, and interpreting disparate data sources to derive meaningful insights remains a significant bottleneck [1, 4]. Users and organizations increasingly struggle to harness this growing body of information in a timely and flexible manner that can adapt to evolving questions and contexts.

Data Integration No Longer Scales. Traditional data management approaches to query disparate data sources depend heavily on integrating data into centralized databases governed by a carefully designed global schema. This process typically involves complex extract, transform, load pipelines, and semantic alignment to create a unified, queryable view of all sources. While such methodologies have worked well when data sources grew slowly, they begin to break as the scale, heterogeneity, and dynamism of data sources increase [9]. Integrations are costly to build and maintain, and once data is integrated, the ability to pose new or unexpected queries becomes constrained by the rigidity of the predefined schema.

Querying without Integration. Traditional query systems rely on predefined schemas and tightly integrated data, making them rigid and expensive to maintain when new data becomes available

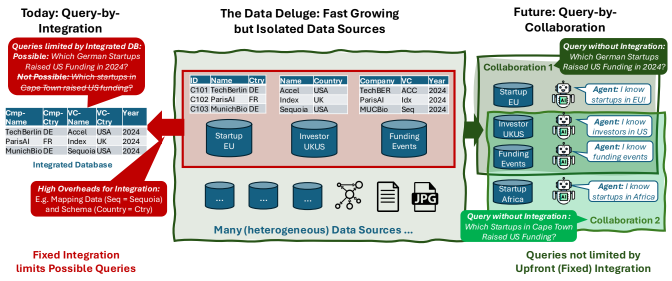

Today: Query-by-Integration
Queries limited by Integrated DB:
Possible: Which German Startups Raised US Funding in 2024?
~~Not Possible: Which startups in Cape Town raised US funding?~~
Cmp-Name | Cmp-Ctry | VC-Name | VC-Ctry | Year
TechBerlin DE Accel USA 2024
ParisAI FR Index UK 2024
MunichBio DE Sequoia USA 2024
Integrated Database
High Overheads for Integration:
E.g. Mapping Data (Seq = Sequoia) and Schema (Country = Ctry)
Fixed Integration limits Possible Queries
The Data Deluge: Fast Growing but Isolated Data Sources
ID | Name | Ctry
C101 TechBerlin DE
C102 ParisAI FR
C103 MunichBio DE
Name | Country
Accel USA
Index UK
Sequoia USA
Company | VC | Year
TechBER ACC 2024
ParisAI Idx 2024
MUCBio Seq 2024
Startup EU
Investor UKUS
Funding Events
...
JPG
Many (heterogeneous) Data Sources ...
Future: Query-by-Collaboration
Query without Integration: Which German Startups Raised US Funding in 2024?
Collaboration 1
Agent: I know startups in EU!
Agent: I know investors in US
Agent: I know funding events
Startup Africa
Agent: I know startups in Africa
Query without Integration : Which Startups in Cape Town Raised US Funding?
Collaboration 2
Queries not limited by Upfront (Fixed) Integration

Figure 1: Data Integration vs. Query Collaboration. Traditional query-by-integration (left) requires integration using rigid global schemas, which become bottlenecks as the number of data sources and heterogeneity grow. In contrast, query-by-collaboration (right) enables schema-agnostic, on-the-fly querying through autonomous data agents that collaboratively resolve queries without global integration. In our vision, query answers thus emerge through interaction; agents negotiate responsibilities and exchange intermediate results to collectively synthesize a final answer.

[2] (see Figure 1, left). To keep pace with the constant availability of new data sources and make them available for querying, we need systems that support *on-the-fly* data composition — where queries are dynamically resolved by discovering and combining data from multiple sources in their native forms, without integration [9, 11] (see Figure 1, right). As a query language, we believe that natural language (NL) should be used, as users want to query data without knowing its concrete organization [3, 5]. In fact, in different sources, data might use highly heterogeneous representations, and between data sources, no exact relationships, such as foreign-key/primary-key relationships, might exist, making the query complex.

Our Vision: Autonomous Data Agent Collaboration. In this paper, we thus suggest a vision of a network of autonomous data agents. Each agent encapsulates a data source and advertises what it knows and how it can respond to queries. When a user submits a query, agents interpret the request locally, produce partial answers, and collaborate through communication. While other ideas to leverage multi-agent systems for data processing exist, such as [10, 12], we are the first to design a framework for collaborative query answering over disparate data sources. In our vision, agents broadcast their capabilities, negotiate responsibilities, and exchange intermediate results to collectively produce a final answer. To understand the benefits of our approach, let us look at a simple example.

A Simple Example. Imagine a user asking, "Which German startups raised funding from US investors in 2024?" Following our vision, assume no single database contains all the required information, as shown in Figure 1 (collaboration 1, right). Instead, multiple autonomous agents are involved: one manages company registration data, including company countries; another maintains funding event records linking companies and investors over time; and a third holds investor profiles with country information. Through a dynamic, iterative exchange of partial results, the agents collaboratively construct the final answer. Importantly, this process does not rely on a predefined schema and fixed query plans but instead emerges from a flexible, distributed collaboration protocol. That way, other queries, such as in Figure 1 (collaboration 2, right), which involve other subsets of agents, can be supported without additional overhead.

The Query Collaboration Protocol. To operationalize this vision, we introduce the *Query Collaboration Protocol (QCP)*. Just as the Model-Context-Protocol (MCP) makes tool integration declarative for agents, QCP makes data querying declarative for agents. To be more precise, QCP defines (1) how data agents expose their local data and querying capabilities in a declarative manner, and (2) how agents can collaborate for query answering by executing partial queries and sharing intermediate results without a global schema. We envision that, based on QCP, we can build new query systems where agents responsible for individual data sources collaboratively answer user queries. However, designing such systems requires a new approach, which is very different from classical databases.

A First Prototype: `QCP-DB`. As a proof of concept to show the feasibility of our vision, we developed `QCP-DB`, a prototype system that applies this collaborative querying approach to structured, tabular data sources. In `QCP-DB`, each agent wraps a small independent relational database and exposes metadata describing its data and query capabilities. When a user query (in natural language) is routed to an agent, the agent interprets the NL query based on its local data and rewrites it into a partial SQL query that expresses the portion of the question it can answer using its local data. This local interpretation ensures that agents contribute partial results only within their scope while collectively resolving the global query. Although `QCP-DB` currently focuses on structured data, we believe that the architecture we present in this paper generalizes to other data modalities—like graphs or even text and images—so long as agents can declare their data and query capabilities.

Contributions and Outline. In Section 2, we introduce the concept of *Autonomous Data Agent Collaboration*, outlining a QCP-based system design and key technical challenges. Section 3 presents our prototype, `QCP-DB`, which implements QCP for structured relational data. Section 4 reports initial results using a modified version of the BIRD benchmark, showing that `QCP-DB` achieves accuracy comparable to querying centralized integrated data, without integration overhead. Finally, Section 5 outlines a roadmap toward realizing the full vision.

2 Autonomous Data Agent Collaboration

This section presents the overall vision of the Query Collaboration Protocol (QCP) and how to enable QCP-based query systems. For this, we first introduce the core components of a QCP-based query system and describe how they interact to resolve queries. Afterwards, we outline the key challenges involved in enabling such systems. We explain in more detail how we implemented each component in Section 3 for our prototype `QCP-DB`.

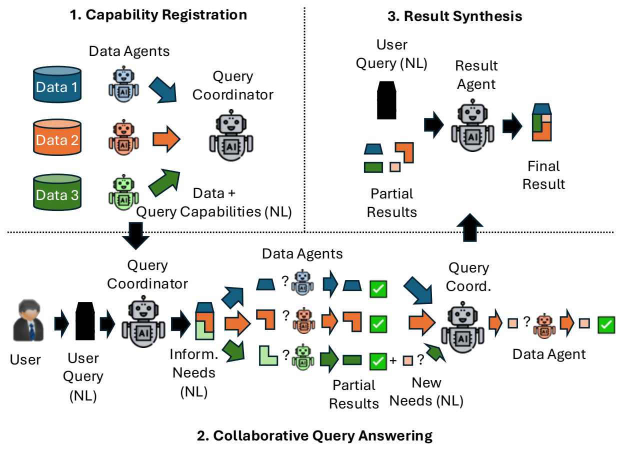


Figure 2: System Sketch for QCP-based Query Processing. (1) Agents register their data and query capabilities with the coordinator using natural language (NL) descriptions. (2) To answer a user's NL query, the coordinator decomposes it into information needs (NL-based sub-queries) and assigns them to agents. Each involved agent receives an NL sub-query, determines whether it can answer it, translates it into a local query on its data source (e.g., using SQL), and returns a partial result. If an agent can only provide part of the answer, it generates a new information need (e.g., green agent) and sends it to the coordinator, who forwards it to another capable agent. (3) A result agent collects all partial answers and synthesizes the final response.

2.1 System Sketch

In Figure 2, we sketch the main components of a QCP-based query processing system, which we describe next.

Query Coordinator. Data agents register dynamically with the coordinator, advertising their capabilities (see Figure 2, upper left). An agent exposes a capability interface that describes metadata (i.e., what the source contains in terms of a natural language description), as well as the query capabilities (e.g., SQL or SPARQL) to the coordinator, which acts on this information. The coordinator is a specialized agent that receives the initial user query in natural language and orchestrates the collaboration process. When a query arrives (Figure 2, bottom), the coordinator identifies candidate agents based on these capabilities and begins routing query fragments accordingly. Although this centralized role simplifies initial coordination, future variants may distribute this function to multiple agents akin to routing in the internet.

Autonomous Data Agents. The most important components in a QCP-based query processing system are the autonomous data agents. We envision each data agent to encapsulate a local data source—such as a relational database, a graph database, or even a vector store for images and text. Importantly, instead of acting as a passive data endpoint, each agent is capable of interpreting natural language input queries that they receive from the coordinator (see Figure 2, bottom), and based on this input query, agents generate local queries over their data, and execute those queries to address the user's complete question or part of it. Alternatively, agents might only be partially able to answer an input query and then submit new information needs back into the system.

Answer Synthesis. A last important function in QCP-based query systems is the generation of the final answer for the user, which is composed by aggregating partial results produced by individual agents. This component is critical for producing coherent results from heterogeneous and distributed data sources. This can be done by a dedicated result agent who collects and integrates the outputs. Other alternatives for result synthesis could also be distributed mechanisms such as voting, scoring, or confidence-based aggregation. In our current prototype, a centralized result agent summarizes all partial responses into a single answer (see Figure 2, upper right).

2.2 Challenges

While the high-level architecture of a QCP-based query processing system is conceptually simple, realizing it in practice involves a number of challenges related to coordination, optimization, and result synthesis.

Query Decomposition & Routing. Because agents are autonomous and encapsulate internal data and logic, the coordinator lacks full visibility into their internals. Decomposing a natural language query into agent-specific sub-tasks and routing them effectively requires new strategies. In particular, an initial decomposition might not be fine-grained enough, or a data agent might discover that a decomposition from the coordinator asks for information that it does not have locally. For handling this, we introduce a *natural language query stack* abstraction that allows the coordinator and agents to collaboratively manage the process, which we outline in detail in the next section.

Query Optimization. Another open question is how query answering can be optimized in settings where we do not have a fixed query plan that defines the steps needed upfront, and data can be heterogeneous. Overall, multi-agent query answering requires minimizing both the number of agent interactions and, at the same time, increasing result quality. One important aspect to optimize is to decide which agents to select and in which order to ask them to answer a user query, which has a strong influence on overall runtime and accuracy. One naive idea could be to simply broadcast the user question to all agents in a network, which then answer whatever portion they can. However, this would be costly and inaccurate, particularly for larger agent networks. As part of this paper, we thus study initial ideas that aim to select and order data agents for query answering.

Data Fragmentation & Semantic Misalignment. Finally, another major challenge is that the nature of data distribution across agents is very different from classical partitioned databases, where data distribution follows a well-known partitioning function and all partitions use a unified representation of the schema and data. Instead, in QCP-based query systems we envision, agents hold independent slices of information, and these are typically not well aligned or normalized in any principled way. In fact, one agent may list company headquarters using English city names ("Geneva"), while another uses French city names ("Genève") or an abbreviation ("GE"). Moreover, explicit links between data sources, such as foreign keys, are missing. As such, joins between distinct data sources must be inferred using potentially noisy attributes. These challenges need to be addressed and affect all phases, including query execution (e.g., when translating incoming user queries to match the data), but also during result synthesis, as we outline in the next section.

3 The `QCP-DB` Prototype

We took a first step towards realizing the QCP vision through `QCP-DB`, a prototype that shows the possibilities for disparate relational data sources. The remainder of this section outlines the prototype's main ideas and implementation decisions alongside Figure 3. We also publish the code of `QCP-DB`[1], which includes all the details such as the exact prompts used for each component as well as the data for reproducing our initial results in Section 4.

3.1 Query Coordinator

The main task of the coordinator is the coordination of collaborative query answering. In our prototype, the coordinator is an LLM-based agent with a strong reasoning model such as OpenAI o3. Next, we describe the individual functions implemented by the coordinator.

Capability Registration. As a first step before agents participate in query answering, the agents register themselves in the coordinator's registry with a metadata description of their local knowledge base as well as their capabilities. While our prototype is for relational databases only, ultimately, we still want to support arbitrary data sources. As such, the metadata each agent sends is a concise natural language description of their schema, as well as its query capability (i.e., SQL). An example of the registration information as we implement it in `QCP-DB` is shown in Figure 3 ①.

Query Routing. When a user query arrives, the coordinator employs a two-phase routing strategy: (1) selecting agents, and (2) assignment to agents.

Agent Selection. First, the coordinator broadcasts the user query to a subset of registered agents using the information in the metadata registry. Afterwards, agents self-assess their potential contribution based on their local data. This aligns with the general design principle of involving as much local knowledge as possible into LLM decisions [6] as the agents know their local data better than the coordinator does through the registration description it has from each agent. An example of this can be seen in Figure 3 ②a. Here, the 'Tech Companies' data agent (blue) and 'Conferences' data agent (green) decide that they can contribute to solving the given user question "Which tech companies hosted a conference in Europe with over 5000 attendees in 2024?" Agents that decide that they can contribute respond with their database schema using the M-Schema format [8], which includes table names, column names, three distinct values per column, and column descriptions in NL, which are created by each data agent automatically for its local schema and data.

Assignment to Agents. Afterwards, the coordinator analyzes the information from the agents who participate to create an initial query assignment using the Query Stack as shown in Figure 3 ②b. For example, to answer the user question in our example, the coordinator analyzes the previously received database schema of the two out of three agents that decided to contribute in Phase 1 ('Tech Companies' & 'Conferences'). It then creates a plan of sub-questions in NL that need to be solved sequentially to answer the complete question. During this process, the coordinator also

[1]https://github.com/DataManagementLab/A-Vision-for-Autonomous-Data-Agent-Collaboration

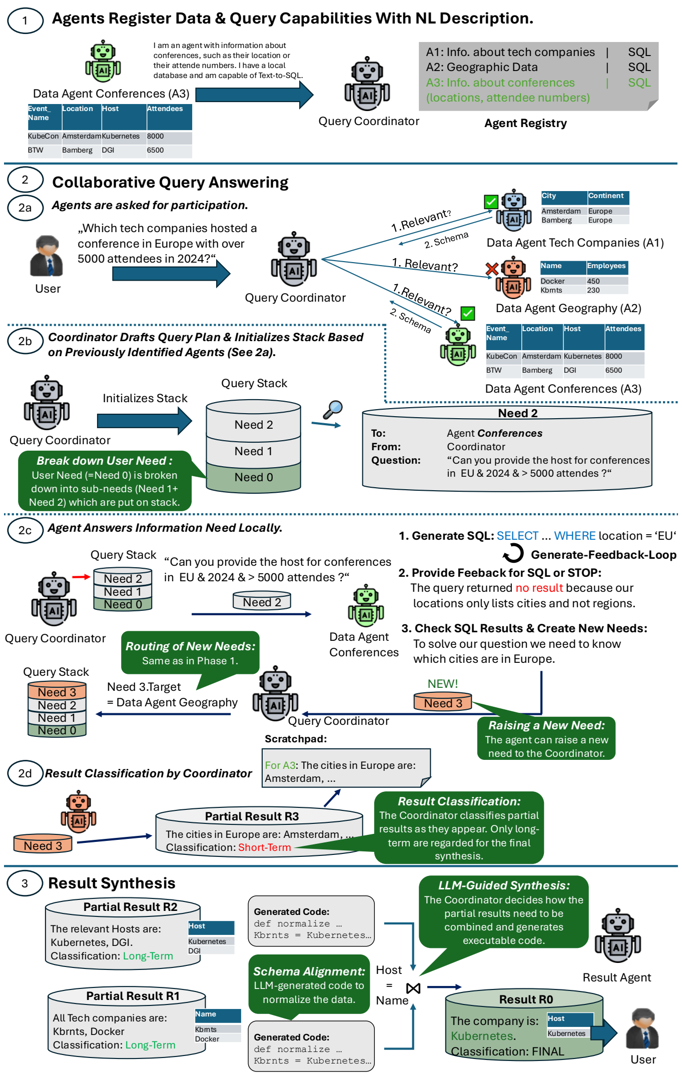

1 Agents Register Data & Query Capabilities With NL Description.
I am an agent with information about conferences, such as their location or their attende numbers. I have a local database and am capable of Text-to-SQL.
Data Agent Conferences (A3)
Event_Name | Location | Host | Attendees
KubeCon | Amsterdam | Kubernetes | 8000
BTW | Bamberg | DGI | 6500
Query Coordinator
A1: Info. about tech companies | SQL
A2: Geographic Data | SQL
A3: Info. about conferences (locations, attendee numbers) | SQL
Agent Registry
2 Collaborative Query Answering
2a Agents are asked for participation.
„Which tech companies hosted a conference in Europe with over 5000 attendees in 2024?"
User
Query Coordinator
1.Relevant?
2. Schema
1. Relevant?
1.Relevant?
2. Schema
City | Continent
Amsterdam | Europe
Bamberg | Europe
Data Agent Tech Companies (A1)
Name | Employees
Docker | 450
Kbrnts | 230
Data Agent Geography (A2)
Event_Name | Location | Host | Attendees
KubeCon | Amsterdam | Kubernetes | 8000
BTW | Bamberg | DGI | 6500
Data Agent Conferences (A3)
2b Coordinator Drafts Query Plan & Initializes Stack Based on Previously Identified Agents (See 2a).
Query Coordinator
Initializes Stack
Query Stack
Need 2
Need 1
Need 0
Break down User Need : User Need (=Need 0) is broken down into sub-needs (Need 1+ Need 2) which are put on stack.
Need 2
To: Agent Conferences
From: Coordinator
Question: "Can you provide the host for conferences in EU & 2024 & > 5000 attendes ?"
2c Agent Answers Information Need Locally.
Query Stack
Need 2
Need 1
Need 0
Query Coordinator
"Can you provide the host for conferences in EU & 2024 & > 5000 attendes ?"
Need 2
Data Agent Conferences
1. Generate SQL: SELECT ... WHERE location = 'EU'
Generate-Feedback-Loop
2. Provide Feeback for SQL or STOP: The query returned no result because our locations only lists cities and not regions.
3. Check SQL Results & Create New Needs: To solve our question we need to know which cities are in Europe.
Routing of New Needs: Same as in Phase 1.
Query Stack
Need 3
Need 2
Need 1
Need 0
Need 3.Target = Data Agent Geography
Query Coordinator
NEW!
Need 3
Raising a New Need: The agent can raise a new need to the Coordinator.
Scratchpad:
For A3: The cities in Europe are: Amsterdam, ...
2d Result Classification by Coordinator
Need 3
Partial Result R3
The cities in Europe are: Amsterdam, ...
Classification: Short-Term
Result Classification: The Coordinator classifies partial results as they appear. Only long-term are regarded for the final synthesis.
3 Result Synthesis
Partial Result R2
The relevant Hosts are: Kubernetes, DGI.
Classification: Long-Term
Host
Kubernetes
DGI
Partial Result R1
All Tech companies are: Kbrnts, Docker
Classification: Long-Term
Name
Kbrnts
Docker
Generated Code: def normalize … Kbrnts = Kubernetes…
Schema Alignment: LLM-generated code to normalize the data.
Generated Code: def normalize … Kbrnts = Kubernetes…
Host = Name ⋈
LLM-Guided Synthesis: The Coordinator decides how the partial results need to be combined and generates executable code.
Result Agent
Result R0
The company is: Kubernetes.
Host
Kubernetes
Classification: FINAL
User

Figure 3: The QCP-DB Prototype for Collaborative Query Processing over Data Agents with Relational Data. ① Agents register dynamically in QCP-DB with a natural language description at the Coordinator. ②ⓐ When a user query arrives, the agents are first asked for participation, and ②ⓑ the Coordinator then creates a stack of information needs that need to be fulfilled sequentially to answer the user question. ②ⓒ Each data agent tries to solve a given need through probing its database with a generate-feedback loop to address the semantic misalignment of needs and data. Furthermore, during the generate-feedback-loop, new information needs are raised to the Coordinator. ②ⓓ Partial results from agents are classified into short-term (intermediate) or as relevant for the final result. ③ A result agent assembles a joint result from partial results marked as relevant for the final result.

assigns one or multiple agents to answer each sub-question. This query decomposition enables the coordinator, together with the data agents, to process complex queries efficiently, with each agent working on the portion of the query best suited to their local data. To order execution and keep track of dependencies (i.e., which sub-questions need to be executed in which order), the coordinator manages a query stack, which we explain next.

Query Stack Details. We implemented the natural language query stack as a stack of so-called NeedNodes. As can be seen in Figure 3 ②ⓑ (Need 2 on the right), a NeedNode contains a sub-question in NL, a list of destination agents, which is initially filled with the set of agents the coordinator thinks can answer the information need, and the originator of the need who requires the answer to the need. All information needs are created by the coordinator and pushed onto the stack for sequential processing (see Figure 3 ②ⓑ), with the NeedNode at the very bottom that contains the user's question, such that it is processed last. Initially, all needs on the stack originate from the coordinator as the initial breakdown of the user question into sub-questions. However, as we see later, additional needs might be discovered during processing by data agents and pushed onto the query stack (see Figure 3 ②ⓒ). More details can be found below in Section 3.2.

3.2 Autonomous Data Agents

In the following, we provide details for the functionalities of the data agents in QCP-DB as outlined in Figure 3. Data agents are responsible for answering sub-questions (i.e., information needs) independently and contributing the partial results to the final answer.

Local Question Interpretation. When an agent receives a natural language information need during query answering, it first interprets this need in the context of its local database as can be seen in Figure 3 ②ⓒ. The idea is that the agent interprets the question locally against its schema and generates SQL queries that are tailored to its specific local database. The difficulty in the translation is that there is likely a mismatch between the incoming information need and how the data is represented in an agent as users are not aware of how data is organized at all. As such, data agents implement a self-fixing generate-feedback-loop as shown at the top right of Figure 3 ②ⓒ, where a SQL query that is generated for the information need is executed by the agent. The agent then examines whether the result answers the question and adjusts the generated SQL query as needed or derives additional needs as we describe next.

Generate-Feedback-Loop. The generate-feedback-loop is a separate LLM call that analyzes each generated SQL using its result to guide query adaptation. It examines three key aspects: (1) whether the query returned expected results, (2) how the local schema represents the requested information by revisiting the column descriptions and a sample of 50 distinct values for each column, and (3) what adjustments to the SQL query might bridge any existing semantic gaps on the value or schema level that the agent did not consider during the initial SQL generation. For example, the agent of Figure 3 ②ⓒ on the top right will not get the intended result with its generated filter for 'EU'. It is the task of the generate-feedback-loop to analyze the schema with its column descriptions and example values closely and then either adapt the query, stop the adaptation if the query answers the need, or decide that additional information is needed before answering is possible.

Raising New Needs. During the generate-feedback-loop, agents might discover that they require additional information to answer the question they currently aim to answer. For example, in Figure 3 ②ⓒ, the agent correctly identifies that it requires a mapping of which cities are in Europe. Importantly, such new needs get checked by the coordinator, and those new needs are added to the Query Stack, which are processed by other agents. For example, in Figure 3 ②c, a new need (Need 3) is generated, which is then routed to the Geography agent (A2). One this new need is answered, the coordinator then routes Need 2 back to the Conference data agent (A3), which can now answer the need "Can you provide the host for conferences in EU ... ?" based on the partial result for Need 3.

Handling Intermediate Results. An interesting point is how intermediate results are transferred between agents. For this, the coordinator decides for each result if it is a short-term result (i.e., it is only relevant as an intermediate result) or if it is relevant for the final result. If it is considered a short-term result it is put into the shared scratchpad, which is forwarded to data agents as needed for answering subsequent needs on the stack. For this, the coordinator decides when processing the next need, which data from the scratchpad is forwarded as part of the context to an agent to answer a need. For example, the coordinator forwards the partial result R3 of Need 3 from the scratchpad, as shown in Figure 3 ②c (bottom), to the Conference data agent to answer Need 2, which is next in the stack.

3.3 Result Synthesis

As data agents return partial results, a result agent receives them and aggregates them into a final result as shown in Figure 3 ③. For merging results from different agents, which might be semantically misaligned, we use a two-phase synthesis process of result alignment and then answer synthesis.

Results Alignment by Code Generation. First, for schema alignment, the result agent uses an LLM to analyze the different column structures of partial results and generate executable Python code that maps disparate column names to a unified schema. For example, it might generate code that maps "corp_name", "company", and "CompanyName" to a single "company_name" column. Second, for value alignment, the result agent generates normalization functions to handle semantic variations in the data. This includes creating code to standardize representations like mapping "TechBerlin GmbH" to "TechBerlin" for joining, or recognizing that "Munich", "München", and "MUC" refer to the same city. The result agent executes this generated alignment code to transform the data before performing joins or unions of partial results. When joining data from complementary data sources, it additionally uses LLM-guided join suggestions to identify semantically equivalent columns across agents—recognizing that Startup EU's "company_name" should join with Funding Events' "Company" field despite the naming difference. This code-generation approach provides the flexibility to handle arbitrary schema and value variations while maintaining the precision needed for accurate data integration.

Final Answer Synthesis. The aggregation process builds incrementally the final result by merging partial results sequentially, one after another, using the approach above. The order in which results are merged is determined by the result agent and is driven by the original user query. Once all needs have been merged, the result agent then synthesizes the final answer. This involves two phases: data processing and presentation. In the processing phase, the result agent determines if additional operations (aggregations, other calculations, sorting) are needed to fully answer the user's question and generates appropriate transformations. In the presentation phase, it analyzes which columns are essential for the answer and removes unnecessary attributes. The result is then shared with the user (e.g., Result R0 in Figure 3 ③).

4 Initial Results

Next, we present the results of our initial experimental study.

Dataset. We evaluate QCP-DB using the development set of the BIRD benchmark [7], which contains 11 databases and 1,534 queries of varying complexity. However, while BIRD provides a comprehensive dataset for testing Text-to-SQL capabilities, each query assumes a single clean integrated database. To create meaningful multi-agent scenarios, we thus fragment each database using three strategies: horizontal (dividing table rows across agents), vertical (distributing related parts of tables to different agents), and random (mixed horizontal & vertical). It is important that all resulting fragments are allocated randomly to agents, which causes complex interaction schemes (see anecdote, Exp 1 & 2). Additionally, we added noise to the data and schema to make collaboration harder (see Exp. 3).

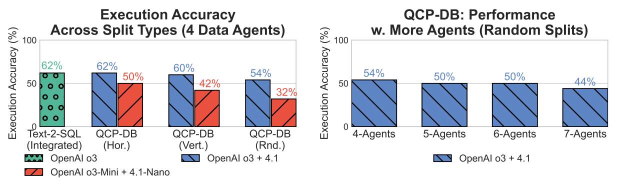
Queries. Initial experiments revealed that many BIRD queries can be answered using only a small amount of data and thus even after partitioning, many queries could be answered by a single partition (i.e., single agent). Therefore, we manually selected a representative sample of 50 queries that specifically challenge multi-agent coordination. Our selection criteria focused on queries of simple and moderate difficulty that require data from multiple agents after fragmentation. We excluded queries marked as "challenging" in BIRD, as these often fail even on centralized Text-to-SQL systems due to their inherent complexity rather than due to data agent collaboration issues.

Baseline. As our upper-bound baseline, we use a single-agent configuration of QCP-DB with access to the complete, unfragmented database. This baseline agent uses core Text-to-SQL components, which we also use for QCP-DB. This includes the M-Schema [8] for schema representation to LLMs, schema-linking, and iterative error correction, ensuring a fair comparison that isolates the effects of fragmentation and coordination. While more sophisticated Text-to-SQL approaches exist, our focus is on evaluating multi-agent coordination rather than advancing Text-to-SQL techniques. As such, our baseline provides a reasonable upper bound for what QCP-DB should achieve if coordination were perfect.

Metric. In line with other Text-to-SQL research, we report execution accuracy (EX) as our primary metric. However, it is important that high execution accuracy is harder to achieve in a system such as QCP-DB due to the heterogeneity of data in the individual agents.

4.1 Exp. 1 & 2: Overall Performance & Scaling

Figure 4a shows how each split type (horizontal, vertical, random) impacts QCP-DB's performance. Figure 4a shows two setups for QCP-DB: The blue bars use OpenAI o3 for the coordinator and GPT-4.1 for the agents. The red bars use the lighter OpenAI o3-Mini for the coordinator and GPT-4.1-Nano for the agents. What we observe is that the accuracy gap widens as we move from horizontal to random fragmentation. Moreover, we see how weaker models struggle when coordination and final result synthesis become harder.



(a) QCP-DB vs. Baseline (green). **(b) Scaling # of Agents.**

Figure 4: Subplot (a) shows that QCP-DB achieves comparable performance to a Text-2-SQL agent that operates on integrated databases when splitting the integrated databases horizontally, vertically, or randomly into 4 pieces, each carried by one agent. Subplot (b) shows that QCP-DB's performance decreases as the integrated databases get split into more pieces, each carried by one agent.

In addition, as a second experiment shown in Figure 4b, we increased the number of random splits from 4 up to 7 per table and distributed data among more agents. In this setup, we use OpenAI o3 for the coordinator and GPT-4.1 for the agents. As can be seen in Figure 4b, the performance with more data agents only decreases slightly from 54% at 4 agents to 44% at 7, indicating that collaboration becomes slightly harder the more agents collaborate, but performance does not significantly drop.

Anecdote. Initially, we assumed that the horizontal splitting would be easily solvable by QCP-DB as it seems to be a rather naive concatenation of data. However, our analysis revealed surprising complexity that challenges this assumption. For example, imagine two tables—`Purchases(PID, ProdID, Amount)` and `Products(ProdID, Category)`—split in half by row number across two agents. The first agent, A_1, keeps purchases (1, 10, 150) and (2, 11, 80) together with product rows (10,electronics) and (11,clothing). The second agent, A_2, holds purchases (3,10,50) and (4, 12, 120) plus product rows (12, toys) and (13, electronics). Suppose a user asks: *"Return the* `Amount` *for every purchase whose product category is* `electronics`*."* Because A_1 can see that product 10 belongs to `electronics`, it can instantly output (1,150). Yet the query is still incomplete: another purchase of the same product is in A_2 (3,50). To avoid losing that row, A_2 must realize that it could contribute important rows even though it cannot locally fulfill the where condition. To contribute, it needs to raise a need for the join key that can substitute the filtering, here the single value {10}, so that it can contribute its own matching purchases. Only after A_2 adds (3,50) is the global answer correct. This small scenario shows why even seemingly obvious horizontal splits are highly challenging.

4.2 Exp. 3: Noise in Data Agents

Another big challenge of QCP-DB is its ability to combine results from data sources with different data and schema representations. We therefore manually applied noise to the data and schema. Our noise includes renaming of columns, changing of data type, using data abbreviations, synonyms, and even translation of data from English into German in some agents. In this experiment, we use the vertical split from Exp. 1 as it requires an alignment of schema and data before executing the union of differently noised sources, which we identified as the most challenging scenario for result alignment. In our evaluation runs with our most powerful setup, we see a strong resilience to noise, with accuracy decreasing only marginally (by 4%). Remarkably, translating data across sources into different languages (German) had no impact on LLM performance in our experiment whatsoever. The 4% performance decrease we observed stemmed from two specific issues: an incorrect SQL generation at the data agent level which led to an incomplete result, and a schema alignment error during final result synthesis that led to an empty final result as a necessary join was not performed.

5 The Road Ahead

With this paper, we presented a first vision towards enabling QCP-based query systems. However, many challenges and questions are still unanswered. One clear next step is to extend QCP-DB beyond relational data sources, and we believe that the basic design principles we implemented in QCP-DB allow us to do so. Moreover, beyond heterogeneity of sources, there are many other challenges and opportunities. Below, we mention two interesting directions in more depth.

Query Answering at Internet-scale. In the long run, we believe that the idea of QCP and QCP-based query systems lays the groundwork for an even larger vision — an open-ended global network, like the Internet, but with autonomous data agents. Ultimately, QCP and QCP-based systems will democratize access to data through open-ended querying, just as the Internet democratized access to global information. At the scale of the Internet, with potentially billions or more data agents, however, many additional challenges arise in discovering relevant agents, decomposing queries efficiently, and minimizing communication and coordination overhead. However, we believe that such flexible agent collaborations already have value at a smaller scale — also to query disparate data sources within companies or across a few companies (e.g., in the medical domain). In this paper, we present an initial step towards this broader vision and show a proof-of-concept that QCP enables autonomous query answering to be possible at all at this smaller scale.

Query Optimization without Plans. With QCP-DB, we have taken initial steps toward query optimization in QCP-based query systems by agent selection and query routing, which reduce runtime and cost compared to naive broadcasting. Yet substantial optimization opportunities remain unexplored. One promising direction is adaptive prompt loading for data agents, where the coordinator dynamically adjusts how agents are queried based on progress. Consider a data agent receiving a request for customer data: in early exploration phases, it might be beneficial to expose all available columns and relationships to help the coordinator discover unexpected join paths. Later, when the query structure crystallizes, the coordinator could switch to more targeted querying of specific columns. Overall, future work must develop new optimization techniques for query collaboration.

Acknowledgments

This work has been supported by the BMBF and the state of Hesse as part of the NHR program and the HMWK cluster project 3AI. It was also partially funded by the LOEWE Spitzenprofessur of the state of Hesse and has benefited from early stages of funding from the Deutsche Forschungsgemeinschaft (DFG) under Germany's Excellence Strategy (EXC-3057); funding will begin in 2026. Additional support for this research was provided by SAP.

References

[1] Xin Luna Dong and Divesh Srivastava. 2013. Big Data Integration. *Proceedings of the VLDB Endowment (PVLDB)* 6, 11 (2013), 1188–1189. Tutorial overview on Big Data Integration.

[2] Zhenzhen Gu, Francesco Corcoglioniti, Davide Lanti, Alessandro Mosca, Guohui Xiao, Jing Xiong, and Diego Calvanese. 2024. A Systematic Overview of Data Federation Systems. *Semantic Web J.* 15, 1 (2024), 107–165. https://doi.org/10.3233/SW-223201

[3] George Katsogiannis-Meimarakis and Georgia Koutrika. 2023. A Survey on Deep Learning Approaches for Text-to-SQL. *The VLDB Journal* 32 (2023), 905–936.

[4] Moe Kayali, Fabian Wenz, Nesime Tatbul, and Çağatay Demiralp. 2025. Mind the Data Gap: Bridging LLMs to Enterprise Data Integration. In *15th Annual Conference on Innovative Data Systems Research (CIDR '25)*. VLDB Endowment, Amsterdam, The Netherlands.

[5] Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024. The Dawn of Natural Language to SQL: Are We Fully Ready? [Experiment, Analysis & Benchmark]. *Proc. VLDB Endow.* 17, 11 (2024), 3318–3331. https://doi.org/10.14778/3681954.3682003

[6] Guoliang Li, Jiayi Wang, Chenyang Zhang, and Jiannan Wang. 2025. Data+AI: LLM4Data and Data4LLM. In *Companion of the 2025 International Conference on Management of Data* (Berlin, Germany) *(SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 837–843. https://doi.org/10.1145/3722212.3725641

[7] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chen-Chuan Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2024. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs. *Advances in Neural Information Processing Systems* 36 (2024).

[8] Yifu Liu, Yin Zhu, Yingqi Gao, Zhiling Luo, Xiaoxia Li, Xiaorong Shi, Yuntao Hong, Jinyang Gao, Yu Li, Bolin Ding, and Jingren Zhou. 2025. XiYan-SQL: A Novel Multi-Generator Framework For Text-to-SQL. (2025). arXiv:2507.04701 [cs.CL] https://arxiv.org/abs/2507.04701

[9] Renee J. Miller. 2018. Open Data Integration. *Proceedings of the VLDB Endowment (PVLDB)* 11, 12 (2018), 2130–2139. https://doi.org/10.14778/3229863.3240491

[10] Zhaoyan Sun, Jiayi Wang, Xinyang Zhao, Jiachi Wang, and Guoliang Li. 2025. Data Agent: A Holistic Architecture for Orchestrating Data+AI Ecosystems. arXiv:2507.01599 [cs.DB] https://arxiv.org/abs/2507.01599

[11] Sepanta Zeighami, Yiming Lin, Shreya Shankar, and Aditya G. Parameswaran. 2025. LLM-Powered Proactive Data Systems. *CoRR* abs/2502.13016 (2025). https://doi.org/10.48550/arXiv.2502.13016

[12] Jun-Peng Zhu, Peng Cai, Kai Xu, Li Li, Yishen Sun, Shuai Zhou, Haihuang Su, Liu Tang, and Qi Liu. 2024. AutoTQA: Towards Autonomous Tabular Question Answering through Multi-Agent Large Language Models. *Proc. VLDB Endow.* 17, 12 (2024), 3920–3933. https://doi.org/10.14778/3685800.3685816